

21世纪全国本科院校电气信息类创新型应用人才培养规划教材

嵌入式系统基础实践教程

主　编　韩　磊

软硬并重，双线展开，不偏不倚，内容全面
案例引入，知识归纳，阅读扩展，扣人心弦
理论为基，实践为要，应用为纲，学以致用





说　明

本书电子版仅提供给高校任课教师使用，如有任课教师需要全本教材浏览或需要本书课件等相关教学资料，请联系北京大学出版社客服，微信手机同号：15600139606，扫下面二维码可直接联系。

由于教材版权所限，仅限任课教师索取，谢谢！



21 世纪全国本科院校电气信息类创新型应用人才培养规划教材

嵌入式系统基础实践教程

主　编　韩　磊

副主编　曹欲晓　陈　飞

参　编　丁宋涛　钱　瑛

北京大学出版社
PEKING UNIVERSITY PRESS

内 容 简 介

本书立足于嵌入式系统基本理论，侧重于基础实践开发，面向实际应用；系统地介绍了嵌入式系统的基本概念、组成、设计原则与方法，从嵌入式系统硬件、嵌入式系统软件、嵌入式系统应用三个层面展开论述。在嵌入式硬件方面，按照 ARM 核到 S3C2410 芯片，再到板级系统的顺序逐步扩展，详细介绍了ARM 核的寄存器文件、工作模式和指令系统，以硬件最小系统为突破口，介绍了 S3C2410 芯片的外设接口及板级扩展方法；在嵌入式软件方面，着重阐述了 C 语言面向嵌入式系统编程的特点和技术要领，介绍了 ARM 软件开发工具，结合嵌入式操作系统的基本原理分析了 μC/OS-II 操作系统的源码，讲述了以μC/OS-II 为操作系统的软件体系结构；最后，以嵌入式工业控制为应用案例，介绍了嵌入式系统开发流程以及 S3C2410 和 μC/OS-II 应用设计方案。

本书融合了嵌入式系统的前导知识，内容丰富，思路清晰，可作为本科、专科院校嵌入式系统入门课程的教学用书，也可供基于 S3C2410 和 μC/OS-II 进行应用开发的广大工作人员学习和参考。

图书在版编目(CIP)数据

嵌入式系统基础实践教程/韩磊主编. —北京：北京大学出版社，2013

(21 世纪全国本科院校电气信息类创新型应用人才培养规划教材)

ISBN 978-7-301-

Ⅰ. ①嵌…　Ⅱ. ①韩…　Ⅲ. ①　　Ⅳ. ①

中国版本图书馆 CIP 数据核字(2012)第　　号

书　　名：嵌入式系统基础实践教程

著作责任者：韩　磊　主编

策 划 编 辑：郑　双

责 任 编 辑：郑　双

标 准 书 号：ISBN 978-7-301-　　-/TP

出 版 发 行：北京大学出版社

地　　址：北京市海淀区成府路 205 号　100871

网　　址：http://www.pup.cn　　新浪官方微博：@北京大学出版社

电 子 信 箱：pup_6@163.com

电　　话：邮购部 62752015　发行部 62750672　编辑部 62750667　出版部 62754962

印　刷　者：

发　行　者：北京大学出版社

经　销　者：新华书店

787 毫米×1092 毫米　16 开本　18 印张　410 千字

2013 年 2 月第 1 版　2013 年　月第 1 次印刷

定　　价：　　元

前　　言

在后 PC 时代，计算将不再局限于传统的 PC 和服务器环境，网络计算和移动计算将很快成为人们日常生活的一部分，并逐渐呈现出普及计算模式。作为普及计算的支撑技术，嵌入式实时系统正逐步应用到越来越多的领域，包括工业控制、军事电子、医疗电子、航空航天、交通、飞行控制、通信、多媒体、办公自动化、实时模拟、虚拟现实、信息家电等领域。

嵌入式系统最初的应用是基于单片机的，大多以可编程控制器的形式出现，具有监测、伺服、设备指示等功能，其通常应用于各类工业控制和飞机、导弹等武器装备中，一般没有操作系统的支持，只能通过汇编语言对系统进行直接控制，运行结束后再清除内存。这些装置虽然已经初步具备了嵌入式的应用特点，但仅仅只是使用 8 位的 CPU 芯片来执行一些单线程的程序，因此严格地说还谈不上“系统”的概念。20 世纪 90 年代后，伴随着网络时代的来临，网络、通信、多媒体技术得以发展，8/16 位单片机在速度和内存容量上已经很难满足这些领域的应用需求。而由于集成电路技术的发展，32 位微处理器价格不断下降，综合竞争能力已可以和 8/16 位单片机媲美。32 位微处理器面向嵌入式系统的高端应用，由于速度快，资源丰富，加上应用本身的复杂性、可靠性要求等，软件的开发一般会需要操作系统平台支持。近些年，嵌入式设备大量涌现，如微波炉、数码照相机、机顶盒、手机、PDA、MP3、各种网络设备等。嵌入式系统开发应用需求越来越大，使嵌入式系统成为继 PC 和 Internet 之后 IT 技术的最热点，而构成嵌入式系统的主流趋势是 32 位嵌入式微处理器加实时多任务操作系统，目前的嵌入式系统往往指的是包含这种资源的系统。

随着嵌入式系统的市场快速增长，嵌入式人才缺口将急剧增大。在 IEEE 计算机协会 2004 年 6 月发布的 Computing Curricula Computer Engineering Report, Ironman Draft 报告中把嵌入式系统课程列为计算机工程学科的领域之一，把软硬件协同设计列为高层次的选修课程。美国科罗拉多州立大学/嵌入式系统认证，课程目录包括实时嵌入式系统导论、嵌入式系统设计和嵌入式系统工程训练课程。美国华盛顿大学嵌入式系统课程名称是嵌入式系统设计导论，它基本包括了前面三门课程的内容。正基于此，国内众多高校、职业技术学院和培训机构纷纷开展嵌入式系统的教学和培训工作。但对于嵌入式系统这一跨学科、软硬件集成、与业界需求密切相关的综合性系统来讲，要在短期内建立起一套完整的、科学的、系统的教学体系绝非易事。

嵌入式系统是嵌入到对象体系中的专用计算机系统。嵌入式系统本质上是一个专用计算机系统，包括硬件、软件和固件等方面的知识。因此，学习嵌入式系统需要掌握全面的基础知识，例如，硬件方面，只了解处理器的寄存器、工作模式是不够的，还应理解存储器及存储映射、寻址方式和调试接口，掌握具体处理器芯片的外围设备、接口技术与硬件设计，等等；软件方面，单纯了解操作系统的工作原理、体系结构、API 调用和应用程序开发也是不够的，开发者还要关注操作系统的移植和引导启动、地址映射、驱动程序开发

等复杂的细节问题。面对嵌入式系统庞大的知识体系，不同专业的学生往往因为背景知识匮乏而显得力不从心。计算机科学与技术专业的学生由数据结构、编译原理、操作系统等专业核心课程做支撑，学习嵌入式软件开发的难度不是很大，但是对于硬件系统的比较、选择、理解、分析和设计，感觉很吃力；而电子信息科学与技术专业的学生，情况却相反，他们强于硬件分析设计，弱在嵌入式系统软件开发。这就形成了嵌入式系统的全面知识需求与学生背景知识不足之间的矛盾，该矛盾普遍存在于嵌入式系统教学过程中。

现行的多数嵌入式系统教材往往忽视了嵌入式系统知识体系的全面性和系统性，或片面强调硬件，或重点突出软件，忽略对前导知识的融会贯通，不是最合适的嵌入式系统入门读物。本书着眼于嵌入式系统的构建过程，主抓嵌入式硬件、嵌入式软件两条主线，以低廉的处理器和简单的操作系统作为分析对象，面向应用，突出基础实践，不乏对基础知识的总结以及对前沿技术的展望，逐步带领初学者入门嵌入式系统技术。

本书分 3 篇，共 10 章，具体内容安排如下：

第一篇　硬件篇，主要包括第 1～4 章。主要介绍嵌入式系统的基本概念、组成、发展、应用，ARM 核的特点、体系结构以及指令系统，S3C2410 芯片的接口和板级扩展方法。

第二篇　软件篇，主要包括第 5～9 章。主要介绍面向嵌入式系统环境的 C 语言编程，ARM 软件开发平台，嵌入式操作系统基本原理，然后以 μC/OS-II 操作系统为例，分析其任务管理和内存管理的基本方法，构建了以 μC/OS-II 为核心的软件体系。

第三篇　应用篇，即第 10 章。以嵌入式工业控制器为例，介绍了嵌入式系统开发的基本流程，然后选用 S3C2410 芯片和 μC/OS-II 操作系统构建工业控制器，介绍了多个接口的软硬件设计方法。

本书的第 1～3 章由陈飞编写，第 5、8、9 章由曹欲晓编写，第 4、6、7、10 章由韩磊编写，最后由韩磊通稿。丁宋涛、钱瑛参与了部分章节的编写和修改，在些表示感谢。

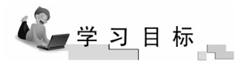
由于作者水平有限，书中难免存在不足之处，敬请广大读者批评指正。

编者

目　　录

第 1 章

嵌入式系统概述



了解嵌入式系统基本概念和特点；
了解嵌入式系统的软硬件体系结构；
了解嵌入式微处理器特点及当前主流的嵌入式微处理器；
了解嵌入式系统的发展现状及趋势。

知 识 结 构

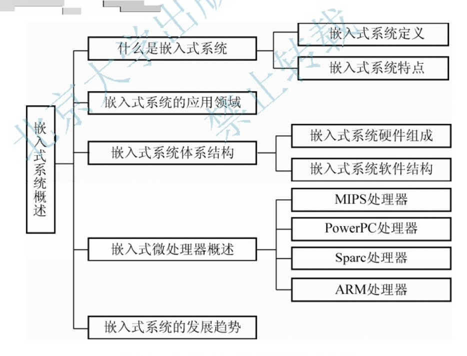
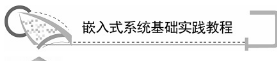


图 1.1 嵌入式系统基本概念知识结构图





导入案例

在介绍嵌入式系统的概念之前，先介绍一种大家都很熟悉的嵌入式产品——智能手机。



图 1.2　iPhone 4 和 iPhone 4S 智能手机

随着移动通信技术的快速发展，手机的功能越来越多，现在的手机已经不只是用于语音通信的设备，而是集成短信、彩信、视频、摄像、游戏、上网、移动办公等多功能的嵌入式产品，这种集成多功能的手机就被称为智能手机。其中，最具有代表性的智能手机，属苹果公司的 iPhone 系列手机，iPhone 3GS、iPhone 4、iPhone 4S，如今已经风靡全球，而 iPhone 5 也已经上市。其中 iPhone 4 和 iPhone 4S 是目前使用最广的苹果手机，以其时尚的外观、高清晰的屏幕、优越的处理器性能以及数不胜数的应用程序，得到广大消费者的热衷。图 1.2 即为 iPhone 4 和 iPhone 4S 的正面外观图。表 1-1 为 iPhone 4S 和 iPhone 4 基本配置。

表 1-1　iPhone 4S 和 iPhone 4 基本配置

产品名称	iPhone 4S	iPhone 4
主屏尺寸	3.5 英寸	3.5 英寸
主屏材质	IPS	IPS
操作系统	iOS 5	iOS 5
核心数	双核	单核
CPU 型号	苹果 A5	苹果 A4
CPU 频率	800MHz	800MHz
内置存储	16GB/32GB/64GB	8GB/16GB/32GB
机身内存	512MB RAM	512MB RAM
电池容量	1420mAh	1420mAh
摄像头像素	前：30 万，后：800 万	前：30 万，后：500 万
WiFi	802.11 b/g/n(2.4GHz)	802.11 b/g/n(2.4GHz)
蓝牙	蓝牙 4.0	蓝牙 2.1
传感器	加速传感器、数字罗盘、陀螺仪	加速传感器、数字罗盘、陀螺仪

iPhone 4 和 iPhone 4S 手机的功能主要有以下几方面。

基本功能：通话功能、铃声设置、通讯录管理、短信功能、邮件收发、中英文输入、内置游戏、移动办公、时间管理；

数据应用功能：蓝牙、上网、USB 接口、无线通信(Wi-Fi)、GPS 定位；
拍照功能：摄像、拍照；
多媒体娱乐：SNS 服务(QQ、微博、飞信、MSN)、多媒体、音乐播放、视频播放、图片处理。

前面给出了一个典型的嵌入式产品的例子，嵌入式系统在实际生活中的应用非常广泛，在日常生活中无处不在。本章将着重讲解嵌入式系统的定义、特点、体系结构等内容。

1.1　什么是嵌入式系统

以微处理器为核心的微型计算机以其小型、价廉、高可靠性的特点，将其嵌入到一个对象体系中，可以实现对对象体系的智能化控制。例如，将微型计算机经电气加固、机械加固，并配置各种外围接口电路，安装到大型舰船中构成自动驾驶仪或轮机状态监测系统。一辆豪华的汽车可能就装配了 70 个以上的微处理器，它们分布在汽车控制系统的众多部件当中。计算机便失去了原来的形态与通用的计算机功能。因此，为了区别于原有的通用计算机系统，把嵌入到对象体系中，实现对象体系智能化控制的计算机，称做嵌入式计算机系统，也就是嵌入式系统。

1. 嵌入式系统定义

目前对于嵌入式系统没有一种准确的定义，下面给出两种比较常见的定义。

第一种，电气和电子工程师协会(IEEE)对嵌入式系统的定义为“用于控制、监视或者辅助操作机器和设备的装置”。可以看出，嵌入式系统是一种装置，是计算机软件和硬件的综合体，还可以涵盖机电等附属装置。

第二种，业界目前普遍采用的定义方式：嵌入式系统是以应用为中心、以计算机技术为基础、软件硬件可裁剪，适应应用系统对功能、可靠性、成本、体积、功耗严格要求的专用计算机系统。它一般由嵌入式微处理器、外围硬件设备、嵌入式操作系统以及用户的应用程序等四个部分组成，用于实现对其他设备的控制、监视或管理等功能。

2. 嵌入式系统特点

与通用的计算机系统不同，嵌入式系统通常具有以下特点。

(1) 专用性

通用计算机可以同时满足多种不同的功能，例如，可以用它来观看视频、听音乐，同时还可以用来开发应用程序。但嵌入式系统只能完成某些特定目的的任务，是面向特定应用，大多工作在特定用户群设计的系统中。

(2) 系统精简

嵌入式系统的软硬件都必须高效设计，在保证系统稳定、安全、可靠的基础上进行量体裁衣，去除冗余。力争用较少的资源实现较高的性能。一方面降低应用成本，另一方面也可以提供系统的安全可靠。在满足应用需求的前提下达到最精简的配置。

(3) 低功耗

有很多的嵌入式系统对象都是一些小型应用系统，如手机、PDA、MP3、数码照相机

等，这些设备不可能像通用计算机一样配置容量较大的电源，也无法配备各种不同的散热片或风扇进行系统散热。低功耗一直是嵌入式系统追求的目标。因此在设计时，有严格的功耗预算，处理器大部分时间都必须工作在低功耗的睡眠模式下，只有在需要处理任务时，才被唤醒。当然也是为了降低系统的功耗，嵌入式系统中的软件一般不存储于磁盘等载体中，而都固化在存储器芯片或单片系统的存储器中。

(4) 实时性

嵌入式系统主要用来对宿主对象进行控制，在很多使用场合，如生产过程控制、传输通信、数据采集、军事设备、航空航天等，都对嵌入式系统有或多或少的实时性要求。大家所熟知的火星探测器上使用的操作系统其实就是一个实时性很高的嵌入式操作系统，上面使用的操作系统就是美国风河系统公司(Wind River System)的 VxWorks 操作系统。现在发展越来越快的 GPS 车辆实时监控系统中同样也对时序和稳定性有一定的需求。车辆移动端的控制器要根据 GPS 的秒信号与整个系统做时钟同步，从而实现移动端数据的分时按时间片向数据中心上报。

(5) 可靠性

可靠性是嵌入式系统一个非常重要的指标，因为工作环境往往比较恶劣，如电磁干扰、高温高湿、静电干扰等，而且嵌入式设备通常都需要在无人值守的场合长时间稳定运行，例如，危险性高的工业环境中，内嵌有嵌入式系统的仪器仪表中，在人烟稀少的气象检测系统中以及为侦察敌方行动的小型智能装置中等。有些嵌入式系统所承担的任务涉及产品质量、人身设备安全、国家机密等，例如，航空航天控制系统一旦发生故障，则可能发生灾难性的后果。所以与普通的计算机系统相比，嵌入式系统对可靠性的要求极高。

(6) 技术融合

嵌入式系统是将先进的计算机技术、半导体技术以及电子技术与各个行业的具体应用相结合的产物，这一点就决定了它必然是一个技术密集、资金密集、高度分散、不断创新的知识集成系统。通用计算机行业中，占整个计算机行业 90%的 PC 产业，绝大部分采用的是 Intel 的 x86 体系结构，而芯片厂商则集中在 Intel、AMD、Cyrix 等几家公司，操作系统方面更是被 Microsoft 占据垄断地位。但这样的情况却不会在嵌入式系统领域出现。这是一个分散的，充满竞争、机遇与创新的工业，没有哪个公司的操作系统和处理器能够垄断市场。

(7) 开发工具和环境

通常嵌入式系统本身是不具备自主开发能力的，即在系统设计完成以后，用户通常不能对其中的程序功能进行修改，因此必须有一套开发工具和环境才能对其进行开发。这些工具和环境一般是基于通用计算机上的软、硬件设备以及各种逻辑分析仪、混合信号示波器等。

1.2 嵌入式系统的应用领域

由于嵌入式系统具有体积、性能、功耗、可靠性等方面的突出优势，目前已经广泛应用于工业控制、军事国防、航空航天、消费电子、信息家电、网络通信等领域。可以说，我们就生活在一个嵌入式的世界，各种电子表、电话、手机、音乐播放器、智能电视、机

顶盒、洗衣机、电饭锅、微波炉都有嵌入式系统的存在。图 1.3 所示为嵌入式系统的应用领域，随着嵌入式技术的不断发展，其应用的前景将更加广阔。

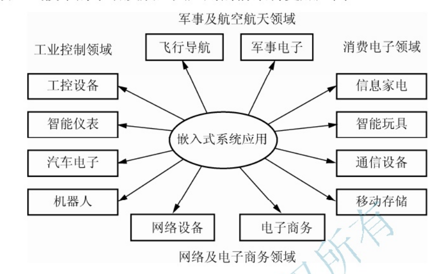


图 1.3　嵌入式系统的应用领域

1. 工业控制

基于嵌入式芯片的工业自动化设备将获得长足的发展，目前已经有大量的 8 位、16 位、32 位嵌入式微控制器在应用中，如工业过程控制、数字机床、电力系统、电网安全、电网设备监测、石油化工系统。就传统的工业控制产品而言，低端型采用的往往是 8 位单片机。但是随着技术的发展，32 位、64 位的处理器逐渐成为工业控制设备的核心，在未来几年内必将获得长足的发展。图 1.4 所示为工业用无线数据采集器，应用于水利、环保、电力等行业，提供高速稳定数据采集和传输功能。



图 1.4　无线数据采集器

2. 军事电子与航空航天

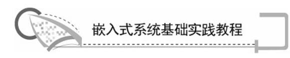
嵌入式系统在军事和航空领域上的应用体现在军事侦察、飞行控制、导弹控制、后勤保障现代化和战场系统网络化等方面。图 1.5 所示为嵌入式系统在导弹控制中的应用，实现对导弹状态的控制和发射。



图 1.5　导弹控制

3. 通信与网络设备

通信系统以及通信网络设备，包括交换机、机顶盒、路由器、调制解调器等。图 1.6 所示为嵌入式系统在高端无线路由器上的应用，传输速度高，无线信号覆盖范围广。



4. 信息家电

信息家电将成为嵌入式系统最大的应用领域，冰箱、空调等的网络化、智能化将引领人们的生活步入一个崭新的空间。即使不在家里，也可以通过电话线、网络进行远程控制。在这些设备中，嵌入式系统将大有用武之地。图 1.7 所示为苹果公司生产的 iPad 2 平板电脑，具有精美的外观和优质的图像处理功能，且具备办公、娱乐等多媒体功能。



图 1.6　高端无线路由器



图 1.7　iPad 平板电脑

5. 家庭智能管理系统



图 1.8　无线点菜器

该系统包括水、电、煤气表的远程自动抄表，安全防火、防盗系统等，其中嵌有的专用控制芯片将代替传统的人工检查，并实现更高、更准确和更安全的性能。目前在服务领域，如图 1.8 所示无线远程点菜器已经体现了嵌入式系统的优势。

6. 汽车电子和交通管理

在车辆控制和导航、流量控制、信息监测与汽车服务方面，嵌入式系统技术已经获得了广泛的应用，内嵌 GPS 模块，GSM 模块的移动定位终端已经在各种运输行业获得了成功的使用。目前 GPS 设备已经从尖端产品进入了普通百姓的家庭，其购买费用只需要几千元。图 1.9 所示为嵌入式系统应用于车辆控制系统。

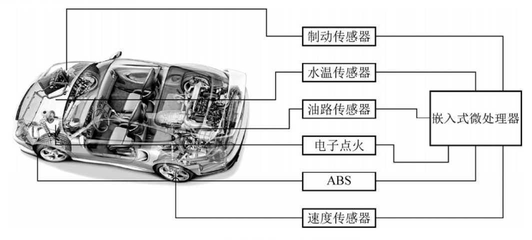


图 1.9　汽车的嵌入式系统控制



7. POS 网络及电子商务

公共交通无接触智能卡(Contactless Smartcard, CSC)发行系统、公共电话卡发行系统、自动售货机、各种智能 ATM 终端将全面走入人们的生活。

8. 环境工程与自然

水文资料实时监测，防洪体系及水土质量监测、堤坝安全，地震监测网，实时气象信息网，水源和空气污染监测。在很多环境恶劣、地况复杂的地区，嵌入式系统将实现无人监测。

9. 机器人

嵌入式芯片的发展将使机器人在微型化、高智能方面的优势更加明显，同时会大幅度降低机器人的价格，使其在工业领域和服务领域获得更广泛的应用。图 1.10 所示为智能机器人瓦力，靠自身动力和控制能力就能完成特定的工作。



图 1.10　瓦力机器人

随着计算机技术和信息技术的不断发展，嵌入式系统的应用范围将越来越广泛，涉及人类生活的各个方面，关系也将越来越紧密，所以，开发和探讨嵌入式系统有着十分重要的意义。

1.3　嵌入式系统体系结构

1.3.1　嵌入式系统硬件组成

图 1.11 所示为一个典型的嵌入式硬件系统组成，以 32 位的 ARM 结构嵌入式微处理器为中心，由存储器、I/O 设备、通信模块以及电源模块等必要的辅助接口组成。其中，嵌入式微处理器为整个系统的核心，决定了系统的功能和应用领域。外围设备则根据实际需求和成本进行裁剪和定制。

1. 嵌入式微处理器

嵌入式硬件系统的核心是嵌入式微处理器，嵌入式微处理器与普通计算机处理器的设计与原理是相似的。最大的不同在于嵌入式微处理器大多工作在为特定用户群所专用设计的系统中，一方面只保留与实际应用相关的功能硬件，去除其他冗余部分，另一方面可以将许多由板卡完成的功能集成在芯片内部。因此，嵌入式微处理器通常体积较小，但具有很高的稳定性和可靠性，功耗低，对环境(温度、湿度、电磁干扰等)的适应能力强。

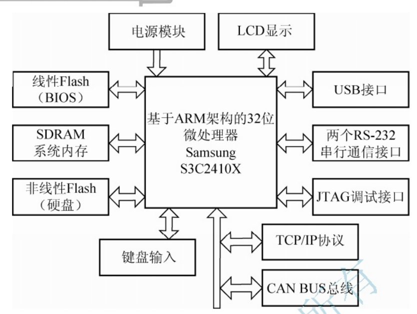


图 1.11　基于三星 S3C2410X 嵌入式微处理器的嵌入式硬件系统

在早期开发的嵌入式系统中，大多采用的是单处理器的架构，随着计算机技术和信息技术的快速发展，现在的嵌入式系统可能采用的是多处理器或多核处理器的硬件架构。例如，智能手机除了满足传统的语音通信的同时，还必须提供稳定和高质量的多媒体需求，传统的单处理器方案不能满足并行任务的处理要求。如苹果 iPhone 4S 智能手机除了采用双核的 A4 核心处理器，还配置有专门的图像处理器，以及通信协议处理器。

2. 存储系统

存储器是构成嵌入式系统的重要组成部分，用于存放系统运行的数据和程序。嵌入式系统的存储系统功能与通用计算机的存储系统功能并无明显的差异。不同的嵌入式系统需要根据实际应用的需求和成本，选择不同的存储技术和存储设备。在实际的嵌入式系统中，往往采用分级的方法来设计整个存储系统，例如，一个四级的存储系统，包含寄存器、高速缓存、内存和辅助存储器。存储级别越高，存取速度越快，而存储容量则越小。

寄存器是最高一级的存储器。在嵌入式系统中，寄存器组一般是微处理器内含的。有些待使用的数据或者运行的中间结果可以暂存在这些寄存器中。微处理器对芯片内的寄存器读写的速度很快，一般在一个时钟周期内完成。从总体上来说，设置一系列寄存器是为了尽可能减少微处理器直接从外部取数的次数，但是，由于寄存器组集成在芯片内部，受芯片面积和集成度的限制，寄存器的数量不可能做得很多。例如，ARM 微处理器一共有 37 个寄存器，其访问时间则一般为几纳秒。

第二级存储器是高速缓冲存储器(Cache)。Cache 是一种小型、快速的存储器。存放的是最近一段时间微处理器使用最多的程序代码和数据。在需要进行数据读取操作时，微处理器尽可能地从 Cache 中读取数据，而不是从内存中读取，这样就大大改善了系统的性能，提高了微处理器和内存之间的数据传输速率。Cache 的主要目标：减小存储器(如内存和辅助存储器)给微处理器内核造成的存储器访问瓶颈，使处理速度更快，实时性更强。

第三级是主存储器即内存。运行的程序和数据都存放在内存中。它可以位于微处理器



的内部或外部，其容量为 256KB~1GB，根据具体的应用而定，一般片内存储器容量小、速度快，片外存储器容量大。

常用作主存的存储器有以下两类。

- ROM 类：NOR Flash、EPROM 和 PROM 等。
- RAM 类：SRAM、DRAM 和 SDRAM 等。

其中 NOR Flash 凭借其可擦写次数多、存储速度快、存储容量大、价格便宜等优点，在嵌入式领域内得到了广泛应用。

第四级为辅助存储器，这种存储器容量大，但是存取速度比内存要慢得多，主要用来存放大数据量的程序代码或信息。嵌入式系统中常用的外存有硬盘、NAND Flash、CF 卡、MMC 和 SD 卡等。

并不是每个嵌入式系统都必须有这种分级的存储结构，而应当根据系统的性能要求和所选定的处理器功能来设计存储系统。例如，对于采用微控制器较小的系统，其自带的存储器就有可能满足系统要求；而对于 16 位、32 位或 32 位以上的微处理器组成的系统，随着系统性能的提高，存储子系统变得更加复杂，一般都包含了全部四级存储器，甚至更多级别的存储器，如网络存储器、磁盘阵列、分布式文件系统等。

3．通用设备接口和 I/O 接口

嵌入式系统和外界交互需要一定形式的通用设备接口，如 A/D(模/数转换接口)、D/A(数/模转换接口)、I/O 等，外设通过和片外其他设备或传感器的连接来实现微处理器的输入/输出功能。每个外设通常都只有单一的功能，它可以在芯片外也可以内置芯片中。外设的种类很多，可从一个简单的串行通信设备到非常复杂的 802.11 无线设备。目前嵌入式系统中常用的通用设备接口有 A/D、D/A，I/O 接口有 RS-232 接口(串行通信接口)、Ethernet(以太网接口)、USB(通用串行总线接口)、音频接口、VGA 视频输出接口、I2C(现场总线)、SPI(串行外围设备接口)和 IrDA(红外线接口)等。

1.3.2　嵌入式系统软件结构

当设计一个简单的应用程序时，可以不使用操作系统(Operating System，OS)，但在设计一个较为复杂的程序时，可能就需要一个操作系统来管理和控制内存、多任务、周边资源等。依据系统提供的程序界面来编写应用程序，可大大减少应用程序员的负担。

对于使用操作系统的嵌入式系统来说，嵌入式系统软件结构一般包含四个层面：设备驱动层、操作系统层、应用程序接口 API 层和应用程序。如图 1.12 所示为四层的嵌入式软件结构。由于硬件电路的可裁剪性和嵌入式本身的特点，其软件部分也是可裁剪的。对于功能简单，仅包括应用程序的嵌入式系统，一般不使用操作系统，仅有应用程序和设备驱动程序。现代高性能嵌入式系统的应用越来越广泛，操作系统的使用成为必然的发展趋势。

1．驱动层程序

驱动层程序是嵌入式系统中不可缺少的重要组成部分，使用任何外部设备都需要有相应的驱动程序的支持，它为上层软件提供了设备的操作接口。上层软件不用关心设备的具

体内部操作细节，只需调用驱动程序提供的接口。一般包括硬件抽象层 HAL、板级支持包 BSP 和设备驱动程序。

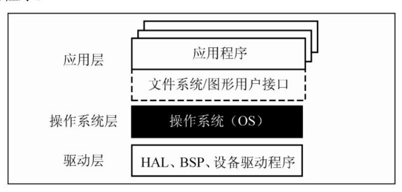


图 1.12　嵌入式系统软件架构

(1) 硬件抽象层

硬件抽象层(Hardware Abstraction Layer，HAL)是位于操作系统内核与硬件电路之间的接口层，其目的在于将硬件抽象化。也就是说，可通过程序来控制所有硬件电路(如 CPU、I/O、Memory 等)的操作。这样就使得系统的设备驱动程序与硬件设备无关，从而大大提高了系统的可移植性。从软、硬件测试的角度来看，软、硬件的测试工作可分别基于硬件抽象层来完成，从而使软、硬件测试工作的并行进行成为可能。在定义硬件抽象层时，需要规定统一的软、硬件接口标准，其设计工作需要基于系统需求，代码编写工作可由对硬件比较熟悉的人员完成。硬件抽象层一般应包括相关硬件的初始化、数据的输入/输出、硬件设备的配置等功能。

(2) 板级支持包

板级支持包(Board Support Package，BSP)是介于主板硬件和操作系统中的驱动程序层之间的一层，一般认为它属于操作系统的一部分，主要是实现对操作系统的支持，为上层的驱动程序提供访问硬件设备寄存器的函数包，使之能够更好地运行于硬件主板。BSP 是相对于操作系统而言的，不同的操作系统对应于不同定义形式的 BSP。例如，VxWorks 的 BSP 和 Linux 的 BSP 相对于某一 CPU 而言，尽管实现的功能可能完全一样，但写法和接口定义却完全不同。BSP 一定要按照该系统 BSP 的定义形式来编程(BSP 的编程过程大多数是在某一个成型的 BSP 模板上进行修改的)，这样才能与上层操作系统保持正确的接口，良好地支持上层操作系统。板级支持包的功能主要体现在两个方面：一是系统启动时，完成对硬件的初始化；二是为驱动程序提供访问硬件的函数接口。

(3) 设备驱动程序

系统中安装设备后，只有在安装相应的设备驱动程序之后才能使用，驱动程序为上层软件提供设备的操作接口。上层软件只需调用驱动程序提供的接口，而不用理会设备的具体内部操作。驱动程序的实现直接影响系统的性能。驱动程序不仅要实现设备的基本功能函数(如初始化、中断响应、发送、接收等)，而且要实现设备的基本功能。因为设备在使用过程中还会出现各种各样的差错，所以好的驱动程序还应该具有完备的错误处理函数。

2. 操作系统

操作系统是计算机系统的管理和控制中心，组织和管理系统资源，包括硬件、软件及数据资源，为其他软件提供支持等。对于使用操作系统的嵌入式系统而言，操作系统一般以内核映像的形式下载到目标系统中。以 μCLinux 为例，在系统开发完成之后，将整个操作系统部分做成内核映像文件，与文件系统一起传送到目标系统中；然后通过 BootLoader 指定地址运行 μCLinux 内核，启动已经下载好的 μCLinux 系统；再通过操作系统解开文件系统，运行应用程序。整个嵌入式系统与通用操作系统类似，功能比不带有操作系统的嵌入式系统强大了很多。

内核中必需的基本部件是进程管理、进程间通信、内存管理等，其他部件如文件系统、驱动程序、网络协议等都可根据用户要求进行配置，并以相关的方式实现。

常用的嵌入式操作系统有以下几种。

(1) Linux 操作系统

Linux 操作系统类似于 UNIX，是一种免费的、源代码完全开放的、符合 POSIX 标准规范的操作系统。由于 Linux 的系统界面和编程接口与 UNIX 相似，所以 UNIX 程序员可以很容易地从 UNIX 环境转移到 Linux 环境中来。Linux 拥有现代操作系统所具有的内容：真正的抢先式多任务处理，支持多用户、内存保护、虚拟内存，支持对称多处理机 SMP(Symmetric Multi-Processing)，符合 POSIX 标准，支持 TCP/IP，支持 32/64 位 CPU。嵌入式 Linux 版本众多，如支持硬实时的 Linux(RTLinux/RTAI)、Embedix、Blue Cat Linux 和 Hard Hat Linux 等。

(2) Windows CE 操作系统

Microsoft 公司 Windows CE 是针对有限资源的平台而设计的多线程、完整优先权、多任务的操作系统，但它不是一个硬实时操作系统。高度模块化是 Windows CE 的一个显著的特性，这一特性有利于它对从掌上电脑到专用工业控制器的用户电子设备进行定制。Windows CE 嵌入式操作系统最大的特点是能提供与 PC 类似的图形界面和主要的应用程序。Windows CE 嵌入式操作系统界面显示的大多是在 Windows 操作系统中出现的标准部件，包括桌面、任务栏、窗口、图标和控件等。这样，只要是对 PC 上的 Windows 操作系统熟悉的用户，就可很快地使用基于 Windows CE 嵌入式操作系统的嵌入式设备。

(3) μC/OS-II 操作系统

μC/OS-II 操作系统是一个可裁剪、源码开放、结构小巧、抢先式的实时多任务内核，主要面向中小型嵌入式系统，具有执行效率高、占用空间小、可移植性强、实时性能优良和可扩展性能强等特点。μC/OS-II 中最多可支持 64 个任务，分别对应优先级 0～63，其中 0 为最高优先级。实时内核在任何时候都是运行就绪状态的最高优先级的任务，是真正的实时操作系统。μC/OS-II 最大程度地使用 ANSI C 语言开发，现已成功移植到近 40 多种处理器体系上。

(4) VxWorks 实时操作系统

VxWorks 操作系统是美国 WindRiver 公司于 1983 年设计开发的一种商用嵌入式实时操作系统(RTOS)。良好的持续能力、高性能的内核以及友好的用户开发环境，在嵌入式实

时操作系统领域占据一席之地。它以其良好的可靠性和卓越的实时性被广泛地应用在通信、军事、航空、航天等高精尖技术及实时性要求极高的领域中，如卫星通信、军事演习、弹道制导、飞机导航等。

3. 应用程序接口 API

应用程序接口(Application Programming Interface，API)是一系列复杂的函数、消息和结构的集合体。嵌入式操作系统的 API 与一般操作系统下的 API 在功能、含义及知识体系上完全一致。可以这样理解 API：在计算机系统中有很多可通过硬件或外部设备执行的功能，这些功能的执行可通过计算机操作系统或硬件预留的标准指令调用，而软件人员不需要为每种功能重新编制程序，只需按系统或某些硬件事先提供的 API 调用即可完成功能的执行。在操作系统中提供标准的 API 函数，可加快用户应用程序的开发，统一应用程序的开发标准，同时也为操作系统版本的升级带来了方便。在 API 函数中，提供的大量的常用模块，可大大简化用户应用程序的编写。

4. 应用程序

操作系统调度应用程序是为处理某个特定任务的代码。一个应用程序完成一个处理任务，操作系统控制整个运行环境。一个嵌入式系统可以只有一个活动的应用，也可以几个应用同时运行。

实际的嵌入式系统应用软件建立在系统的主任务(Main Task)基础之上。用户应用程序主要通过调用系统的 API 函数对系统进行操作，完成用户应用功能的开发。在用户的应用程序中，可创建用户自己的任务。任务之间的协调主要依赖系统的消息队列。

1.4 嵌入式微处理器概述

嵌入式微处理器(Micro Processor Unit，MPU)是由通用计算机中的微处理器演变而来的。不同的是，在实际嵌入式应用中，只保留和嵌入式应用紧密相关的功能部件，去除其他的冗余功能部分，这样就以最低的功耗和资源实现嵌入式应用的特殊要求。和通用控制计算机相比，嵌入式微处理器具有体积小、质量轻、成本低、可靠性高的优点。嵌入式微处理器一般具有以下特点：

- 对实时和多任务有很强的支持能力。处理器内部具有精确的振荡电路、丰富的定时器资源，能完成多任务并且有较短的中断响应时间，从而使内部的代码的执行时间减少到最低限度；
- 具有功能很强的存储区保护功能。这是由于嵌入式系统的软件结构已模块化，而为了避免在软件模块之间出现错误的交叉作用，需要设计强大的存储区保护功能，同时也有利于软件诊断；
- 可扩展的处理器结构。一般在处理器内部都留有很多的扩展接口，以方便迅速扩展出满足应用的高性能的嵌入式微处理器；
- 嵌入式微处理器的功耗必须很低，尤其是用于便携式的无线及移动的计算和通信设备中，依靠电池供电的嵌入式系统更是如此，功耗只能为 mW 甚至 μW 级。

目前比较有影响的主流嵌入式微处理器产品有 MIPS 公司的 MIPS、IBM 公司的 PowerPC、Sun 公司的 Sparc、ARM 公司的 ARM 系列，下面将分别介绍。

1. MIPS 处理器

MIPS 技术公司是一家设计制造高性能、高档次及嵌入式 32 位和 64 位处理器的厂商。在 RISC 处理器方面占有重要地位。1984 年，MIPS 计算机公司成立。1992 年，SGI 收购了 MIPS 计算机公司。1998 年，MIPS 脱离 SGI，成为 MIPS 技术公司。

MIPS 的意思是“无内部互锁流水级的微处理器”(Microprocessor without Interlocked Piped Stages)，最早是在 20 世纪 80 年代初期由美国斯坦福大学 Hennessy 教授领导的研究小组研制出来的。1986 年推出 R2000 处理器，1988 年推出 R3000 处理器，1991 年推出第一款 64 位商用微处理器 R4000，之后，又陆续推出 R8000(1994 年)、R10000(1996 年)和 R12000(1997 年)等型号。之后，MIPS 公司的战略发生变化，把重点放在嵌入式系统。1999 年，MIPS 公司发布 MIPS32 和 MIPS64 位架构标准，为未来 MIPS 处理器的开发奠定了基础。新的架构集成了所有原来 MIPS 指令集，并且增加了许多更强大的功能。MIPS 公司陆续开发了高性能、低功耗的 32 位处理器内核(core)MIPS32 4Kc 与高性能 64 位处理器内核 MIPS64 5Kc。2000 年，MIPS 公司发布了针对 MIPS32 4Kc 的新版本以及未来 64 位 MIPS64 20Kc 处理器内核。MIPS 公司新近推出的 MIPS32 24K 微架构，适合支持各种新一代嵌入式设计，例如，视讯转换器与 DTV 等需要相当高的系统效能与应用设定弹性的数字消费性电子产品。此外，24K 微架构能符合各种新兴的服务趋势，为宽频存取以及还在不断发展的网络基础设施、通信协议提供软件可编程的弹性。

在嵌入式方面，MIPS 系列微处理器是目前仅次于 ARM 的用得最多的处理器之一(1999 年以前，MIPS 是世界上用得最多的处理器)，其应用领域覆盖游戏机、路由器、激光打印机、掌上电脑等各个方面。MIPS 的系统结构及设计理念比较先进，在设计理念上 MIPS 强调软、硬件协同提供性能，同时简化硬件设计。

2. PowerPC 处理器

PowerPC 架构的特点是可伸缩性好，方便灵活。PowerPC 处理器品种很多，既有通用的处理器，又有嵌入式控制器和内核，应用范围非常广泛，从高端的工作站、服务器到桌面计算机系统，从消费电子产品到大型通信设备，无所不包。

处理器芯片主要型号是 PowerPC750。它于 1997 年研制成功，其最高工作频率可达 500MHz，采用先进的铜线技术。该处理器有许多品种，以适合各种不同的系统，包括 IBM 小型机、苹果电脑和其他系统。

嵌入式的 PowerPC405(主频最高为 266MHz)和 PowerPC440(主频最高为 550MHz)处理器内核可用于各种 SoC 设计上，在电信、金融和其他许多行业具有广泛的应用。

3. Sparc 处理器

Sun 公司以性能优秀的工作站闻名，这些工作站的心脏全都是采用 Sun 公司自己研发的 Sparc 芯片。根据 Sun 公司未来的发展规划，在 64 位 UltraSparc 处理器方面，主要有三个系列。首先是可扩展式 s 系列，主要用于高性能、易扩展的多处理器系统。目前

UltraSparcIIIs 的频率已达到 750MHz。将推出 UltraSparc IVs 和 UltraSparc Vs 等型号，其中 UltraSparc IVs 的频率为 1GHz，UltraSparc Vs 则为 1.5GHz。其次是集成式 i 系列，它将多种系统功能集成在一个处理器上，为单处理器系统提供了更高的效益。已经推出的 UltraSparc IIIi 的频率达到 700MHz，未来的 UltraSparc IVi 的频率将达到 1GHz。最后是嵌入式 e 系列，它为用户提供理想的性能价格比，嵌入式应用包括瘦客户机、电缆调制解调器和网络接口等。Sun 公司还将推出主频 300MHz、400MHz、500MHz 等版本的处理器。

4. ARM 处理器

ARM(Advanced RISC Machines)系列处理器是 ARM 公司的产品。ARM 公司是业界领先的知识产权供应商。与一般公司不同，ARM 公司只采用 IP(Intelligence Property) 授权的方式允许半导体公司生产基于 ARM 的处理器产品，提供基于 ARM 处理器内核的系统芯片解决方案和技术授权，但 ARM 公司不提供具体的芯片。

ARM 公司设计先进数字产品的核心应用技术，应用领域涉及无线、网络、消费娱乐、影像、汽车电子、安全应用及存储装置等各种嵌入式领域。ARM 提供广泛的产品，包括 16/32 位 RISC 微处理器、数据引擎、三维图形处理器、数字单元库、嵌入式存储器、外设、软件、开发工具以及高速连接产品。

ARM 公司协同众多技术合作伙伴，为业界提供快速、稳定的完整系统解决方案。ARM 的全球合作伙伴主要为半导体和系统伙伴、操作系统伙伴、开发工具伙伴、应用伙伴。ARM 的紧密合作伙伴已发展为 122 家半导体和系统合作伙伴、50 家操作系统合作伙伴、35 家技术共享合作伙伴，并于 2002 年在上海成立了中国全资子公司。

ARM 取得了极大的成功，已形成完整的产业链，世界上几乎所有主要的半导体厂商都从 ARM 公司购买 IP 许可，利用 ARM 核开发面向各种应用的 SoC 芯片。目前 ARM 系列芯片已被广泛应用于移动电话、手持式计算机以及各种各样的嵌入式应用领域，成为世界上销量最大的 32 位微处理器。

ARM 的成功在于它极好的性能以及极低的能耗，使得它能够与高端的 MIPS 和 PowerPC 嵌入式微处理器抗衡。另外，根据市场需求进行功能的扩展，也是 ARM 取得成功的一个重要因素。随着更多厂商的支持和加入，可以预见，在将来一段时间之内，ARM 仍将主宰 32 位嵌入式处理器市场。

基于 ARM 核嵌入式芯片的典型应用：汽车产品，如车上娱乐系统、车上安全装置、自主导航系统等；消费娱乐产品，如数字视频、Internet 终端、交互电视、机顶盒、网络计算机、数字音频播放器、数字音乐板、游戏机等；数字影像产品，如信息家电、数字照相机、数字系统打印机；工业控制产品，如机器人控制、工程机械、冶金控制、化工生产控制等；网络产品，如 PCI 网络接口卡、ADSL 调制解调器、路由器等；安全产品，如电子付费终端、银行系统付费终端、智能卡、32 位 SIM 卡等；存储产品，如 PCI 到 Ultra2 SCSI64 位 RAID 控制器、硬盘控制器；无线产品，如手机、PDA，目前 85%以上的手机是基于 ARM 做的。

基于 ARM 的应用还有许多，以上只是对一些已进入广泛使用的领域进行粗略的概括。从以上应用可看出，以 ARM 为主流的嵌入式控制芯片应用已十分广泛。随着国家经济的



快速发展，经济实力的不断增强，人民生活水平的逐步提高，自动化装备程度的不断提高，嵌入式设备将无孔不入，也将很快渗透到各行各业，而且它往往以人们意想不到的形式存在，为人们的生活、学习和工作提供各种便捷的服务。

1.5 嵌入式系统的发展趋势

嵌入式系统产品正不断渗透到各个行业，随着嵌入式系统应用的不断深入和产业化程度的不断提升，新的应用环境和产业化需求对嵌入式系统提出了更加严格的要求。在新需求的推动下，嵌入式系统不仅需要具有微型化、高实时性等基本特征，还将向高可靠性、自适应性、构件组件化方向发展；支撑开发环境将更加集成化、自动化、人性化；对无线通信和能源管理的功能支持将日益重要。总的来讲，嵌入式系统的发展趋势有多功能、微型化、低功耗、网络化、信息化的特点，未来嵌入式系统的几大发展趋势如下。

1. 嵌入式开发的系统工程

嵌入式开发是一项系统工程，因此要求嵌入式系统厂商不仅要提供嵌入式软硬件系统本身，同时还需要提供强大的硬件开发工具和软件包支持。

目前很多厂商已经充分考虑到这一点，在主推系统的同时，将开发环境也作为重点推广。例如，三星在推广 Arm7，Arm9 芯片的同时还提供开发板和板级支持包(BSP)，而 Window CE 在主推系统时也提供 Embedded VC++作为开发工具，还有 Vxworks 的 Tonado 开发环境，DeltaOS 的 Limda 编译环境等都是这一趋势的典型体现。当然，这也是市场竞争的结果。

2. 网络化、信息化的要求

随着 Internet 技术的成熟、带宽的提高，网络化、信息化的要求日益提高，使得以往单一功能的设备如电话、手机、冰箱、微波炉等功能不再单一，结构更加复杂。

这就要求芯片设计厂商在芯片上集成更多的功能，为了满足应用功能的升级，设计师们一方面采用更强大的嵌入式处理器如 32 位、64 位 RISC 芯片或信号处理器 DSP 增强处理能力，同时增加功能接口，如 USB；扩展总线类型，如 CAN BUS，加强对多媒体、图形等的处理，逐步实施片上系统的概念。软件方面采用实时多任务编程技术和交叉开发工具技术来控制功能复杂性，简化应用程序设计，保障软件质量和缩短开发周期，如 HP。

3. 网络互联成为必然趋势

当前行业内有一种普遍共识，在未来几年内，越来越多的互联网设备将不再以计算机为主导，而会以嵌入式产品的形式出现，也就是我们称之为的“嵌入式互联网设备”。事实上，从网络视频监控系统、智能家庭远程控制到智能电视机顶盒、互联网电视以及数以百万辆拥有互联网接入能力的汽车，嵌入式互联网设备正逐渐从工业级应用渗透到普通消费者的日常生活中。未来的嵌入式设备为了适应网络发展的要求，必然要求硬件上提供各种网络通信接口。传统的单片机对于网络支持不足，而新一代的嵌入式处理器已经开始内嵌网络接口，除了支持 TCP/IP 协议，还有的支持 IEEE 1394、USB、CAN、Bluetooth 或 IrDA 通信接口中的一种或者几种，同时也需要提供相应的通信组网协议软件和物理层驱动

软件。软件方面系统内核支持网络模块，甚至可以在设备上嵌入 Web 浏览器，真正实现随时随地使用各种设备上网。

4. 精简系统内核、算法并降低功耗和软硬件成本

未来的嵌入式产品是软、硬件紧密结合的设备，为了减低功耗和成本，需要设计者尽量精简系统内核，只保留和系统功能紧密相关的软硬件，利用最低的资源实现最适当的功能，这就要求设计者选用最佳的编程模型和不断改进算法，优化编译器性能。因此，既要软件人员有丰富的硬件知识，又需要发展先进嵌入式软件技术，如 Java、Web 和 WAP 等。

5. 提供友好的多媒体人机界面

嵌入式设备能与用户亲密接触，最重要的因素就是它能提供非常友好的用户界面。图像界面，灵活的控制方式，使得人们感觉嵌入式设备就像是一个熟悉的老朋友。这方面的要求使得嵌入式软件设计者必须在图形界面，多媒体技术上痛下苦功。手写文字输入、语音拨号上网、收发电子邮件以及彩色图形、图像都会使使用者获得自由的感受。目前一些先进的 PDA 在显示屏幕上已实现汉字写入、短消息语音发布，但一般的嵌入式设备距离这个要求还有很长的路要走。

6. 多核技术的应用

无所不在的智能必将带来无所不在的计算，大量的图像信息也需要传递给处理器来处理，面对海量数据，单个处理器可能无法在规定的时间完成处理。解决这个问题的关键是引入并行计算技术，可以采用多个执行单元同时处理，这就是处理器的多核技术。因此，在嵌入式微处理器中引入多核技术也是未来嵌入式处理器发展的必然趋势。

1.6 案 例 分 析

本章导入案例中给出的是嵌入式系统在智能手机领域的典型应用，并以最具有代表性的 iPhone 4 和 iPhone 4S 手机为例介绍了智能手机的特点和基本功能。本节根据嵌入式系统的基本概念，介绍智能手机的硬件和软件的总体框架。

1. 硬件架构

典型的智能手机硬件体系结构通常采用双 CPU 的结构，如图 1.13 所示。

主处理器运行开放式操作系统，负责整个系统的控制。从处理器为无线 Modem 部分，主从处理器之间通过串口进行通信。案例中，iPhone 4S 手机的主处理器为基于 Cortex-A9 架构的苹果 A5 双核处理器，最高主频为 1GHz，其性能、功效和功能都达到前所未有的高度。主处理器上含有 LCD(液晶显示器)控制器、SDRAM 和 SDROM 控制器、通用 GPIO 口、SD 卡接口等。无线 Modem 部分作为主处理器的一个外设，本身也是一个独立的系统，内部运行完整的通信协议和独立的电源管理模块，实现无线信号的接收和处理、语音传输和编解码。和音频处理芯片进行通信，构成通话过程的语音通道。

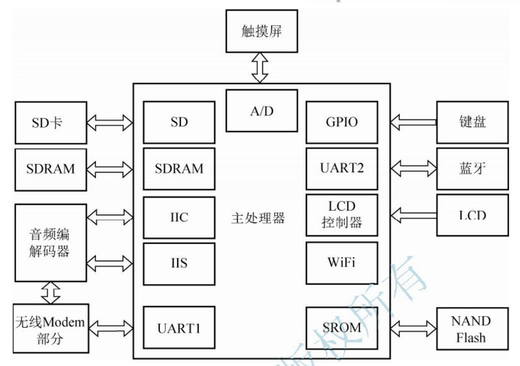


图 1.13　智能手机硬件结构

2. 软件设计

智能手机的软件设计是智能手机系统实现的关键，设计的优劣直接关系到系统的稳定性、可移植性和可扩展性。软件系统层次结构如图 1.14 所示。

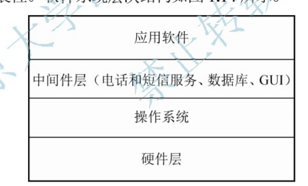


图 1.14　智能手机软件系统层次结构

最上层为应用软件层，包括手机基本应用软件和其他软件，如互联网应用、电话和短消息应用、通讯录、视频软件、图像处理软件和游戏软件等，实现手机的基本功能和其他办公、娱乐功能。

中间件层为智能手机应用程序提供功能支持。例如，包含嵌入式 GUI、嵌入式数据库、电话和短信服务。嵌入式 GUI 实现人机交互，嵌入式数据库统一管理各种数据，电话和短信服务提供智能手机电话和短消息接口支持。

操作系统层包含两个部分，一部分是与底层硬件相关的设备驱动，如串口驱动、LCD 驱动、触摸屏驱动等；另一部分是操作系统内核，实现进程调度、内存管理、进程间通信和文件系统等功能。

本 章 小 结

嵌入式系统是一种嵌入到对象体系中，实现对象体系智能化控制的计算机系统。嵌入式系统已经渗透到人类生活的各个领域，随着嵌入式技术的蓬勃发展，应用的范围将更加广泛。本章主要介绍嵌入式系统的定义、特点以及应用领域，介绍嵌入式系统的硬件和软件架构，以及各个部件的基本功能。

(1) 嵌入式系统的定义和特点：嵌入式系统是以应用为中心、以计算机技术为基础、软件硬件可裁剪，适应应用系统对功能、可靠性、成本、体积、功耗严格要求的专用计算机系统。具有专用性强、可靠性高、低功耗、实时性强等特点。

(2) 嵌入式系统应用领域：嵌入式系统具有体积、性能、功耗、可靠性等方面的突出优势，广泛应用于工业控制、军事国防、航空航天、消费电子、信息家电、网络通信等领域。在日常生活中是无处不在。

(3) 嵌入式系统体系结构：典型的嵌入式硬件系统是以嵌入式处理器为中心，由存储器、I/O 设备、通信模块以及电源模块等必要的辅助接口组成；嵌入式系统软件结构一般包含四个层面：设备驱动层、操作系统层、应用程序接口 API 层和应用程序。

(4) 嵌入式微处理器：嵌入式微处理器是嵌入式系统的核心，是控制、辅助系统运行的硬件单元。目前应用较为广泛的主流嵌入式微处理器有 MIPS 处理器、PowerPC 处理器、Sparc 处理器和 ARM 处理器。

(5) 嵌入式系统发展趋势：嵌入式系统逐渐向多功能、高性能、微型化、低功耗、网络化、信息化的方向发展。


阅读材料

嵌入式系统发展历史

第一个被大家认可的现代嵌入式系统是麻省理工学院仪器研究室的查尔斯•斯塔克•德雷珀开发的阿波罗导航计算机。在两次月球飞行中他们在太空驾驶舱和月球登陆舱都是用了这种惯性导航系统。

在计划刚开始的时候，阿波罗导航计算机被认为是阿波罗计划风险最大的部分。为了减小尺寸和重量，而使用的当时最新的单片集成电路却加大了阿波罗计划的风险。第一款大批量生产的嵌入式系统是1961 年发布的民兵 I 导弹上的 D-17 自动导航控制计算机。它是由独立的晶体管逻辑电路建造的，带有一个作为主内存的硬盘。当民兵 II 导弹在 1966 年开始生产的时候，D-17 由第一次使用大量集成电路的更新计算机所替代。仅仅这个项目就将与非门集成电路模块的价格从每个 1000 美元降低到了每个 3 美元，使集成电路的商用成为可能。

民兵导弹的嵌入式计算机有一个重要的设计特性：它能够在项目后期对制导算法重新编程以获得更高的导弹精度，并且能够使用计算机测试导弹，从而减少测试用电缆和接头的质量。

这些 20 世纪 60 年代的早期应用，使嵌入式系统得到长足发展，它的价格开始下降，同时处理能力和功能也获得了巨大的提高。Intel 4004 是第一款微处理器，它在计算器和其他小型系统中找到了用武之地。但是，它仍然需要外部存储设备和外部支持芯片。1978 年，国家工程制造商协会发布了可编程微控

制器的“标准”，包括几乎所有以计算机为基础的控制器，如单片机、数控设备，以及基于事件的控制器。

随着微控制器和微处理器的价格下降，一些消费性产品用使用微控制器的数字电路取代，如分压计和可变电容这样的昂贵模拟元件成为可能。

到了 20 世纪 80 年代中期，许多以前是外部系统的元件被集成到了处理器芯片中，这种结构的微处理器得到了更广泛的应用。到了 20 世纪 80 年代末期，微处理器已经出现在几乎所有的电子设备中。

集成化的微处理器使得嵌入式系统的应用扩展到传统计算机无法涉足的领域。对多用途和相对低成本的微控制器进行编程，往往可成为各种不同功能的组件。虽然要做到这一点，嵌入式系统比传统的解决方案要复杂，最复杂的是在微控制器本身。但是嵌入式系统很少有额外的元件，大部分设计工作是软件部分。而非物质性的软件不管是建立原型还是测试新修改，相对于硬件来说，都要容易很多的，并且设计和建造一个新的电路不会修改嵌入式处理器。

习　　题

一、判断题

1．嵌入式系统本质上也是属于计算机系统。　　(　　)

2．只有使用了高性能处理器的产品才属于嵌入式设备，而使用单片机的设备不能称为嵌入式设备。　　(　　)

二、问答题

1．什么叫嵌入式系统？具有什么特点？

2．目前嵌入式系统广泛应用于哪些领域？并举出嵌入式系统的应用例子。

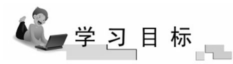
3．什么叫嵌入式微处理器？嵌入式微处理器分为哪几类？

4．目前主流的嵌入式微处理器有哪些？

5．嵌入式系统的发展呈现哪些特征？

第 2 章

嵌入式微处理器核心



学习目标

了解 ARM 处理器指令集体系的发展历程;
熟悉典型 ARM 处理器内核的结构;
理解 ARM 处理器的异常机制、工作状态和运行模式;
掌握 ARM 体系结构的寄存器组织和存储格式。

知识结构

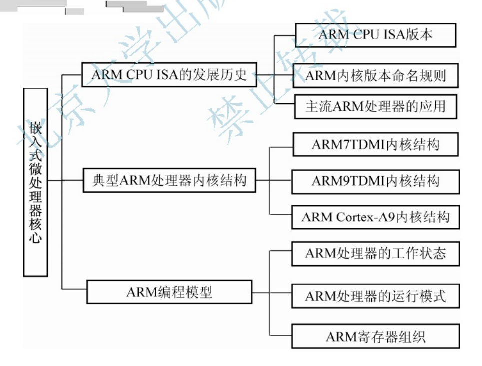


图 2.1　嵌入式微处理器核心知识结构图

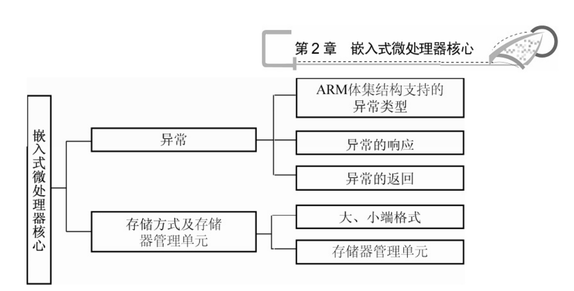


图 2.1　嵌入式微处理器核心知识结构图(续)



导入案例

在第 1 章中，介绍了目前广泛使用的主流嵌入式微处理器。其中以 32 位的嵌入式微处理器为主，而在 32 位嵌入式微处理器中又以 ARM(Advanced RISC Machine)处理器为主，所以本书以 ARM 处理器为核心，介绍嵌入式系统开发的相关软硬件基础知识。

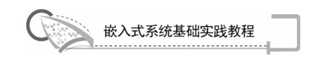
ARM 公司是全球领先的半导体知识产权(IP)提供商，并因此在数字电子产品的开发中处于核心地位。ARM 公司于 1990 年 11 月在英国剑桥成立，公司合作伙伴如图 2.2 所示。



图 2.2　ARM 公司合作伙伴

ARM 的商业模式主要涉及 IP 的设计和许可，而非生产和销售实际的半导体芯片，向合作伙伴网络(包括世界领先的半导体公司和系统公司)授予 IP 许可证。正因为 ARM 的 IP 多种多样以及支持基于 ARM 的解决方案的芯片和软件体系十分庞大，全球领先的原始设备制造商(OEM)都在广泛使用 ARM 技术，应用领域涉及手机、数字机顶盒以及汽车制动系统和网络路由器等。当今，全球 95%以上的手机以及超过四分之一的电子设备都在使用 ARM 技术。

本章着眼于以 ARM 处理器为核心，讲解 ARM CPU ISA 的发展历程，着重介绍典型 ARM 处理器内核的结构，ARM 处理器的编程模型、异常机制和存储格式等。



2.1 ARM CPU ISA 的发展历史

2.1.1 ARM CPU ISA 版本

ARM 处理器的指令集体系结构(Instruction Set Architecture，ISA)从最初的 V1 版本发展到现在，先后出现了 V1、V2、V3、V4、V5、V6、V7 七个主要的版本，在 2011 年 10 月，ARM 公开了最新的 ARM V8 架构的技术细节，如表 2-1 所示。

表 2-1 ARM 体系结构的发展

体系结构	ARM 处理器核
V1	ARM1
V2	ARM2
V2a	ARM2AS、ARM3
V3	ARM6、ARM600、ARM610
V3	ARM7、ARM700、ARM710
V4T	ARM7TDMI、ARM710T、ARM720T、ARM740T
V4	Strong ARM、ARM8、ARM810
V4T	ARM9TDMI、ARM920T、ARM940T
V5TE	ARM9E-S、ARM946E-S、ARM968E-S
V5TE	ARM10TDMI、ARM1020E
V5TEJ	ARM9EJ-S、ARM926EJ-S、ARM7EJ-S、ARM1026EJ-S
V6	ARM1156T2-S、ARM1136J(F)-S、ARM1176JZ(F)-S、ARM11 MPCore
V7	ARM Cotex-A 系列、ARM Cotex-R 系列、ARM Cotex-M 系列
V8	尚未推出处理器

ARM V1～V3 版本的处理器未得到大量应用，而 ARM 处理器的大量广泛应用是从其 V4 版本开始的。下面对 V4 及以上版本的基本特点作简要介绍。

1. ARM V4 版本

版本 4 在前面版本的基础上增加了下列指令：

- 有符号和无符号的半字读取和写入指令；
- 带符号的字节读取和写入指令；
- 增加了处理器的系统模式，在该模式下，使用的是用户模式下的寄存器；
- 版本 4 中明确定义了哪些指令会引起未定义指令异常，不再强制要求与以前的 26 位地址空间兼容。

ARMV4T 版本："T" 扩展表示 Thumb 指令集。ARMV4T 在 ARM V4 的基础上增加了 16 位的 Thumb 指令集。处理器有了 Thumb 状态，并且有了在 ARM 状态和 Thumb 状态切换的指令，处理器在 ARM 状态执行 ARM 指令集，在 Thumb 状态执行 Thumb 指令集。

2. ARM V5 版本

版本 5 在版本 4 的基础上增加或修改以下指令：

- 提高了 T 变种中 ARM/Thumb 混合使用的效率；
- 增加了前导零计数(CLZ)指令，该指令可以使整数除法和中断优先级排队操作更为有效；
- 增加了软件断点(BKPT)指令；
- 为协处理器设计提供了更多可选择的指令；
- 更加严格地定义了乘法指令对条件标志位的影响；
- 带状态切换的子程序调用(BLX)指令。

ARMV5TE 版本："E"扩展表示在通用的 CPU 上扩展了增强的 DSP 指令集，增强的 DSP 指令包括支持饱和算术(Saturated Arithmetic)，并且针对 Audio DSP 应用提高了 70%性能。ARMV5TE 增强了 Thumb 体系，同时提高了 Thumb/ARM 指令交互的性能，从而大大提高了编译器的能力，能够更好地平衡代码量与性能，更好地优化 ARM/Thumb 程序。

ARMV5TEJ 版本："J"扩展表示 Java 加速器 Jazelle 技术，将 Java 的优势与先进的 32 位 RISC 芯片完美结合到一起。与普通的 Java 虚拟机相比，Jazelle 使 Java 代码的运行速度提高了 8 倍，功耗却降低了 80%。

3. ARM V6 版本

版本 6 是 2001 年发布的，其目标是在有效的芯片面积上为嵌入式系统提供更高的性能。ARM V6 包含了 ARMV5TEJ 的所有指令。为了使现有的软件、开发方法、设计技术可再利用，ARM V6 兼容了 ARM V5 的内存管理和异常处理。ARM V6 主要在多媒体处理、存储器管理、多处理器支持、数据处理、异常和中断响应等方面做了改进。

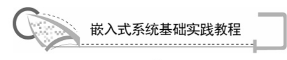
- SIMD(单指令多数据)指令，可使音视频处理能力提高 2～4 倍；
- Thumb-2 新指令集，混合执行 ARM 和 Thumb 代码，可以提供 ARM 指令级别的性能和 Thumb 指令级别的代码密度；
- 混合大小端和非对齐存储访问支持；
- TrustZone 安全技术。

4. ARM V7 版本

版本 7 是 2004 年发布的。全新的 ARM V7 是基于 ARM V6 的，ARM V7 采用了 Thumb-2 技术，体积比 32 位 ARM 代码减小 31%，性能比 16 位 Thumb 代码高出 38%。同时，ARM V7 保持了对已有 ARM 代码的兼容性。

ARM V7 的增加的特性有以下几方面。

- 改进的 Thumb-2 指令集。
- NEON 多媒体技术，将 DSP 和多媒体处理能力提高了近 4 倍。
- 改良的浮点运算，满足下一代 3D 图形、游戏物理应用以及传统嵌入式控制应用的需求。
- ARM V7 定义了三种不同的处理器配置(Processor Profiles)：Profile A 是面向复杂、基于虚拟内存的 OS 和应用的；Profile R 是针对实时系统的；Profile M 是针对低成本应用的优化的微控制器的。



5. ARM V8 版本

2011 年 10 月 31 日，ARM 公司公开了新的 ARM V8 架构的技术细节，这是首款包含 64 位指令集的 ARM 架构。ARM V8 拓展了现有的 32 位 ARM V7 架构，引入了 64 位处理技术，并扩展了虚拟寻址。

ARM V8 架构包含两个执行状态，即 AArch64 和 AArch32。AArch64 执行态针对 64 位处理技术引入了一个全新指令集 A64；而 AArch32 执行态将支持现有的 ARM 指令集。目前 ARM V7 架构的主要特性都将在 ARM V8 架构中得以保留或进一步拓展。

2.1.2 ARM 内核版本命名规则

每一种指令集体系版本可以由多种处理器实现，如表 2-1 所示。ARM 使用如下命名规则来描述一个处理器，命名格式如下：

```
ARM{x}{y}{z}{T}{D}{M}{I}{E}{J}{F}{-S}
```

大括号中的字母是可选的，各个字母的含义如表 2-2 所示。

表 2-2 ARM 内核版本命名规则中的字母含义

字母	含义	字母	含义
x	处理器系列，如 ARM7	I	支持 Embedded ICE，支持嵌入式跟踪调试
y	内存管理/保护单元，如 ARM920T 中的 2 表示有内存管理单元	E	支持增强型 DSP 指令
z	内部含有 Cache	J	支持 Java 加速器 Jazelle
T	支持 Thumb 指令集	F	具备向量浮点单元 VFP
D	支持 JTAG 片上调试	--S	可综合版本
M	支持快速乘法器		

说明：

1) ARM7TDMI 之后的所有 ARM 内核，即使“ARM”标志后没有包含“TDMI”字符，也都默认包含了 TDMI 的功能特性。

2) JTAG 是由 IEEE 1149.1 标准测试访问端口和边界扫描结构来描述的，它是 ARM 用来发送和接收处理器内核与测试仪器之间调试信息的一系列协议。

3) 嵌入式 ICE 宏单元是建立在处理器内部用来设置断点和观察点的调试硬件。

4) 可综合，意味着处理器内核是以源代码形式提供的。这种源代码形式可被编译成一种易于 EDA 工具使用的形式。

2.1.3 主流 ARM 处理器的应用

ARM 处理器的产品系列非常丰富，目前市场上流通使用的 ARM 微处理器主要包括 ARM7、ARM9、ARM11 和 ARM Cortex 系列，以及专门为安全设备设计的 SecurCore 系列。

1. ARM7 系列

ARM7 系列是世界上使用范围最广的 32 位嵌入式处理器系列，是 ARM 面向通用应用的经典处理器系列；具有 170 多个芯片授权使用方，自 1994 年推出以来已销售了 100 多亿台。ARM7 系列具有以下特点：

- 低功耗的 32 位 RISC 处理器，冯 • 诺依曼结构，极低的功耗，适合便携式产品；
- 具有嵌入式 ICE-RT 逻辑，调试开发方便；
- 三级流水线结构，能够提供 0.9MIPS 的三级流水线结构；
- 代码密度高，兼容 16 位的 Thumb 指令集；
- 对操作系统的支持广泛，包括 Windows CE、Linux、Palm OS 等；
- 指令系统与 ARM9 系列兼容，便于用户的产品升级换代；
- 主频最高可达 130MIPS。

主要应用领域包括工业控制、Internet 设备、网络和调制解调器设备、移动电话等多种多媒体和嵌入式应用。

2. ARM9 系列

ARM9 系列微处理器是迄今最受欢迎的 ARM 处理器，是基于 ARM V5 架构的常用处理器系列，在高性能和低功耗特性方面提供最佳的性能。ARM9 系统具有以下特点：

- 五级整数流水线；
- 哈佛体系结构；
- 支持 32 位 ARM 指令集和 16 位 Thumb 指令集；
- 全性能的 MMU，支持 Windows CE、Linux、Palm OS 等多种主流嵌入式操作系统；
- 支持数据 Cache 和指令 Cache，具有更高的指令和数据处理能力。

主要应用包括无线设备、仪器仪表、安全系统、机顶盒、高端打印机、数码照相机和数码摄像机。

3. ARM11 系列

ARM11 处理器系列是基于 ARM V6 架构的高性能处理器，所提供的引擎可用于当前生产领域中的很多智能手机；该系列还广泛用于消费类、家庭和嵌入式应用领域。ARM11 系列具有以下特点：

- 强大的 ARM V6 指令集体系结构；
- ARM /Thumb 指令集可以减少高达 35%的内存带宽和大小需求；
- 用于执行高效嵌入式 Java 的 ARM Jazelle 技术；
- ARM DSP 扩展；
- SIMD(单指令多数据)媒体处理扩展可提供高达两倍的视频处理性能；
- Thumb-2 技术(仅 ARM1156(F)-S)，可提高性能、能效和代码密度。

主要应用包括消费类、无线和汽车信息娱乐、数据存储、图像和嵌入式控制等领域。

4. ARM Cortex 系列

Cortex 系列是 ARM 最新的处理器系列，包括 Cortex-A、Cortex-R 和 Cortex-M 三个系



列，分别用于不同的领域。

Cortex-A 处理器适用于具有高计算要求、运行丰富操作系统以及提供交互媒体和图形体验的应用领域，从最新技术的移动 Internet 必备设备(如手机和超便携的上网本或智能本)到家用网关和下一代数字电视系统等。

Cortex-R 实时处理器为具有严格的实时响应限制的深层嵌入式系统提供高性能计算解决方案。目标应用包括智能手机和基带调制解调器中的移动手机处理；企业系统，如硬盘驱动器、联网和打印；家庭消费性电子产品、机顶盒、数字电视、媒体播放器和照相机；汽车制动系统等。

Cortex-M 系列处理器是 ARM 专门针对需要低功耗和高性能的嵌入式控制市场而开发的，目标应用为智能传感器、人机接口设备、汽车电子和安全气囊、大型家用电器、混合信号设备和微控制器等。

Cortex 系列处理器包括 Cortex-A9、Cortex-M3、Cortex-R4 等。

5．SecurCore 系列

SecurCore 处理器系列是面向高安全性应用的处理器。提供功能强大的 32 位安全解决方案，该系列处理器具有体积小、功耗低、代码密度大和性能高等特点。

- 支持 ARM 指令集和 Thumb 指令集，以提高代码密度和系统性能；
- 采用软内核技术以提供最大限度的灵活性，可防止外部对其进行扫描探测；
- 提供面向智能卡和低成本的存储保护单元 MPU；
- 可以灵活地集成用户自己的安全特性和其他的协处理器。

SecurCore 系列包括 SC000、SC100 和 SC300 处理器。

SecurCore 处理器可用于各种安全应用，如银行业、付费电视、公共交通、电子政务、SIM 卡和证件应用等。

2.2　典型 ARM 处理器内核结构

2.2.1　ARM7TDMI 内核结构

ARM7TDMI 基于 ARM 体系结构 V4T 版本，是目前低端的 ARM 核，具有广泛的应用，其最显著的应用是数字移动电话。基于 ARM7TDMI 的嵌入式处理器有三星公司的 S3C44B0X、飞利浦公司的 LPC2110 等。

ARM7TDMI 是一款 32 位嵌入式 RISC 处理器，能够提供 0.9MIPS 的三级流水线结构，支持 64 位乘法指令，支持片上调试，支持 16 位压缩指令集 Thumb 和 Embedded-ICE 调试单元。ARM7TDMI 内核采用冯 • 诺依曼体系结构，即数据和指令使用同一个存储器，经由同一条总线传输。

ARM7TDMI 内核结构如图 2.3 所示。

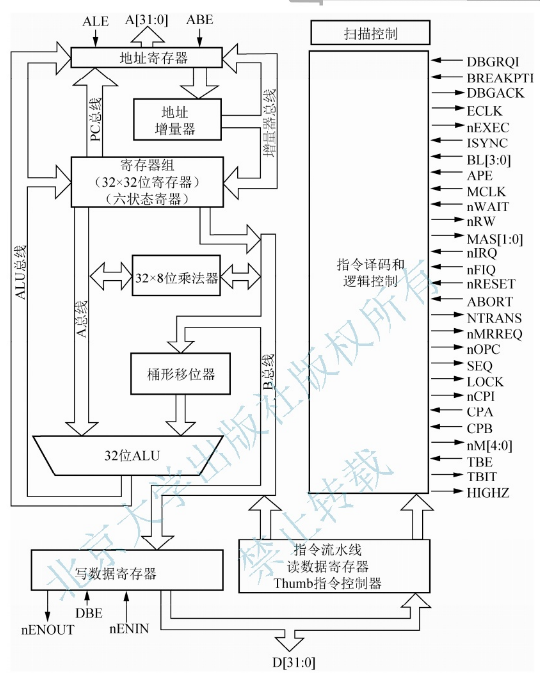


图 2.3　ARM7TDMI 核内部结构

ARM7 系列内核采用了三级流水线的内核结构，三级流水线分别为取指(Fetch)、译码(Decode)、执行(Execute)，如图 2.4 所示。

取指	译码		执行			
指令预取	Thumb → ARM 解压缩	ARM解码 寄存器选择	寄存器读	移位	ALU	寄存器写

图 2.4　ARM7 3 级流水线操作

➢　取指：将指令从存储器中取出，放入指令 Cache 中。

➢ 译码：由译码逻辑单元完成，是将在上一步指令 Cache 中的指令进行解释，告诉 CPU 将如何操作。

➢ 执行：这阶段包括移位操作、读通用寄存器内容、输出结果、写通用寄存器等。

ARM7 的三级指令流水线如图 2.5 所示，图中 PC 为程序计数器。

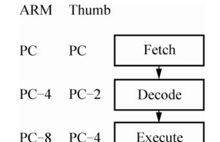


图 2.5　ARM7 的三级指令流水线

注：程序计数器 PC 指向被取指的指令，而不是指向正在执行的指令。

在正常操作的过程中，在执行一条指令的同时，对下一条指令进行译码，并将第三条指令从存储器中取出。

2.2.2　ARM9TDMI 内核结构

ARM9 系列微处理器是迄今最受欢迎的 ARM 处理器，本书使用的硬件平台也是基于 ARM9 处理器的。

ARM9TDMI 核将 ARM7TDMI 的功能显著提高到更高、更强的水平。ARM9TDMI 也支持 Thumb 指令集，并支持片上调试。最显著的区别是流水线从三级增加到 1.1MIPS/MHz 的五级；并且 ARM9 内核采用哈佛体系结构，即使用两个独立的存储模块分别存储指令和数据，每个模块都不允许指令和数据并存，具有独立的地址总线和数据总线。ARM9TDMI 的组织结构如图 2.6 所示。

ARM9TDMI 是一款 32 位嵌入式 RISC 处理器内核。在指令操作上采用五级流水线，其各级操作功能如图 2.7 所示。

➢ 取指：从指令 Cache 中读取指令。

➢ 译码：对指令进行译码，识别出是对哪个寄存器进行操作并从通用寄存器中读取操作数。

➢ 执行：进行 ALU 运算和移位操作，如果是对存储器操作的指令，则在 ALU 中计算出要访问的存储器地址。

➢ 存储器访问：如果是对存储器访问的指令，用来实现数据缓冲功能(通过数据 Cache)。

➢ 寄存器回写：将指令运算或操作结果写回到目标寄存器中。

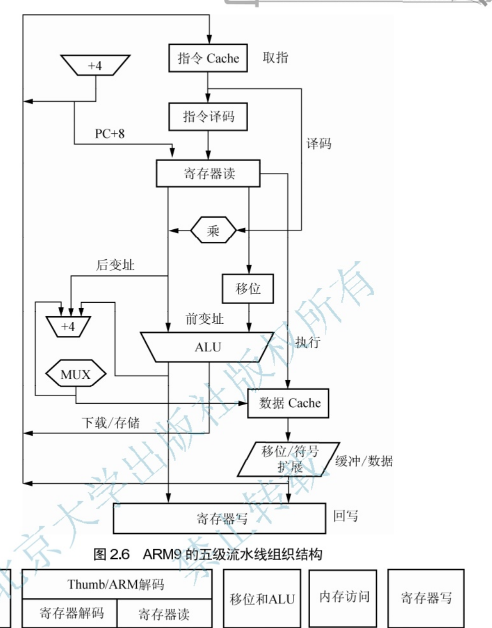


图 2.6　ARM9 的五级流水线组织结构

图 2.7　ARM9 的五级流水线操作

2.2.3　ARM Cortex-A9 内核结构

Cortex-A9 属于 ARM Cortex-A 系列微处理器，是性能最高的 ARM 处理器，可实现受到广泛支持的 ARM V7 体系结构的丰富功能。Cortex-A9 微体系结构既可用于传统的单核处理器，也可用于可伸缩的多核处理器(Cortex-A9 MPCore 多核处理器，最多设计四个处理器内核)。目前市场上如 Samsung Galaxy S II 智能手机、iPhone 4S 都采用了 Cortex-A9 处理器。

Cortex-A9 处理器的设计基于最先进的推测型八级流水线，该流水线具有高效、动态长度、多发射超标量及无序完成等特征。这款处理器的性能、功效和功能均达到了前所未有的水平。根据实测，Cortex-A9 处理器可在 2GHz 主频下保持极低的功耗，完全能够满足消费、网络、企业和移动应用等领域对于 MID(Mobile Internet Device)产品的性能要求。Cortex-A9 单核处理器及多核处理器结构如图 2.8 和图 2.9 所示。

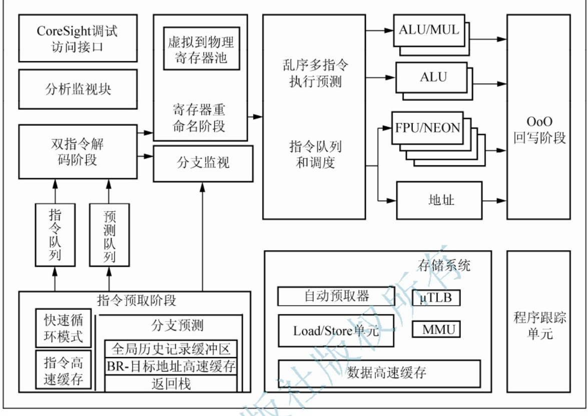


图 2.8 Cortex-A9 单核处理器结构

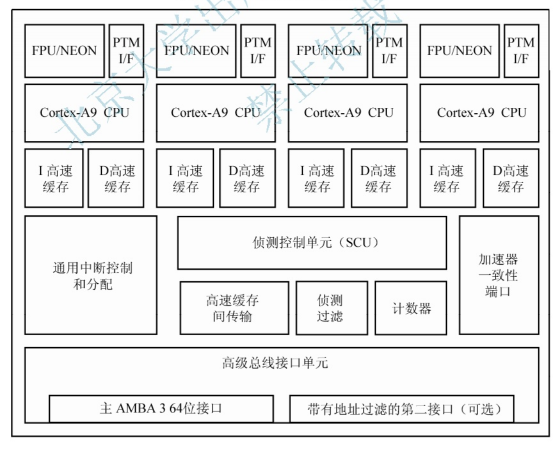


图 2.9 Cortex-A9 多核处理器结构

2.3　ARM 编程模型

在本节，将具体讲解 ARM 处理器编程模型的相关基本概念，包括 ARM 处理器的工作状态、运行模式及 ARM 体系结构的寄存器组织等。

2.3.1　ARM 处理器的工作状态

ARM 处理器有以下两种工作状态。

- ARM：32 位，这种状态下执行字对准的 ARM 指令；
- Thumb：16 位，这种状态下执行半字对准的 Thumb 指令。

在程序执行的过程中，微处理器可以随时在两种状态之间切换。

注：处理器工作状态的转变并不影响处理器运行模式和相应寄存器的内容。

ARM 指令集和 Thumb 指令集均有切换处理器状态的指令。

- 进入 Thumb 状态：执行 BX 指令，并设置操作数寄存器的状态(位[0])为 1，可以使微处理器从 ARM 状态切换到 Thumb 状态；此外，在 Thumb 状态进入异常，处理器会切换到 ARM 状态，当异常处理返回时自动转换到 Thumb 状态。
- 进入 ARM 状态：执行 BX 指令，并设置操作数寄存器的状态(位[0])为 0，可以使微处理器从 Thumb 状态切换到 ARM 状态；此外，处理器进行异常处理时，将 PC 放入异常模式链接寄存器中，从异常向量地址开始执行也可进入 ARM 状态。

2.3.2　ARM 处理器的运行模式

ARM 处理器共有七种运行模式，如表 2-3 所示。

表 2-3　ARM 处理器七种运行模式

处理器模式	说明
用户模式(User，usr)	正常程序执行模式，用于应用程序
快速中断模式(FIQ，fiq)	FIQ 异常响应时，进入此模式，用于支持高速数据传送或通道处理
外部中断模式(IRQ，irq)	IRQ 异常响应时，进入此模式，用于一般中断处理
管理模式(Supervisor，svc)	系统复位和软件中断响应时，进入此模式，用于系统初始化或操作系统功能
数据访问中止模式(Abort，abt)	存储器保护异常处理
未定义指令中止模式(Undefined，und)	未定义指令异常处理
系统模式(System，sys)	运行特权操作系统任务(ARM V4 以上版本)

在以上模式中除了用户模式以外，其他模式都属于特权模式。在特权模式下，程序可以访问所有的系统资源，也可以进行任意的处理器模式切换。除系统模式的其他五种特权模式又称异常模式。

处理器模式可以通过软件控制和异常处理过程进行切换。大多数程序在用户模式下运

行，这时应用程序不能访问一些受系统保护的资源，且应用程序也不能直接进行处理器模式切换。当应用程序发生异常中断时，处理器进入相应的异常模式。在每一种异常模式中都有一组寄存器，供相应的异常处理程序使用，这样可以保证在进入异常模式时，用户模式下的寄存器不被破坏。寄存器的详细说明见 2.3.3 节。

系统模式并不是通过异常进入的，它和用户模式具有完全一样的寄存器。但系统模式属于特权模式，不受用户模式限制。有了这个模式，操作系统要访问用户模式的寄存器就比较方便。同时，操作系统的一些特权任务可以使用系统模式，以访问一些受控资源，而不必担心异常出现时的任务状态变得不可靠。

2.3.3 ARM 寄存器组织

ARM 处理器有 37 个 32 位的寄存器。

- 31 个通用寄存器：包括程序计数器 PC、堆栈指针及其他通用寄存器；
- 6 个状态寄存器。

这些寄存器不能被同时看到，在 ARM 处理器的七种运行模式下，每种模式都有一组与之对应的寄存器组。在任意处理器模式下，可见的(即可编程的)寄存器包括通用寄存器(R0～R14)、一个或两个状态寄存器及程序计数器 PC。在所有的寄存器中，有些是各模式共用的同一个物理寄存器；有些是各模式自己独有的物理寄存器，如表 2-4 所示。

表 2-4　ARM 寄存器组织

寄存器类别	各模式实际访问的物理寄存器						
	用户模式	系统模式	管理模式	中止模式	未定义模式	IRQ 模式	FIQ 模式
通用寄存器	R0						
	R1						
	R2						
	R3						
	R4						
	R5						
	R6						
	R7						
	R8						R8_fiq
	R9						R9_fiq
	R10						R10_fiq
	R11						R11_fiq
	R12						R12_fiq
	R13		R13_svc	R13_abt	R13_und	R13_irq	R13_fiq
	R14		R14_svc	R14_abt	R14_und	R14_irq	R14_fiq
	R15(PC)						
状态寄存器	CPSR						
	无		SPSR_svc	SPSR_abt	SPSR_und	SPSR_irq	SPSR_fiq

1. 通用寄存器

通用寄存器可分成三类：不分组寄存器 R0～R7、分组寄存器 R8～R14、程序计数器 R15，下面分别加以介绍。

(1) 不分组寄存器 R0～R7

R0～R7 是不分组寄存器。这意味着在所有处理器模式下，它们每一个都访问的是同一个物理寄存器，是真正并且在每种状态下都统一的通用寄存器。

不分组寄存器没有被系统用于特别的用途，任何可采用通用寄存器的应用场合都可以使用不分组寄存器，但必须注意对同一寄存器在不同模式下使用时的数据保护。

(2) 分组寄存器 R8～R14

寄存器 R8～R14 为分组寄存器。它们所对应的物理寄存器取决于当前的处理器模式。

寄存器 R8～R12 有两个分组的物理寄存器。一组用于除 FIQ 模式之外的所有模式(R8～R12)；另一组用于 FIQ 模式(R8_fiq～R12_fiq)，这样的结构设计有利于加快 FIQ 的处理速度。

寄存器 R13、R14 分别有六个分组的物理寄存器。一组用于用户和系统模式，其余五组分别用于五种异常模式。

R13 常用作堆栈指针，称做 SP。处理器的每种异常模式下都有自己独立的物理寄存器 R13，所以在用户应用程序的初始化部分，一般要初始化每种模式下的 R13，使其指向该异常向量专用的栈地址。在异常处理程序入口处，将用到的其他寄存器的值保存在堆栈中。返回时，重新将这些值加载到寄存器，起到保护程序现场的作用。

R14 用作子程序链接寄存器(Link Register)，也称为 LR，在结构上有两个特殊功能。

- 在每种处理器模式下，模式自身的 R14 用于保存子程序返回地址。当使用 BL 或 BLX 指令调用子程序时，R14 被设置成子程序返回地址。子程序返回通过将 R14 复制到程序计数器 PC 来实现，如执行“MOV PC，LR”或者“BX LR”两条指令之一。还有一种方式如下。

在子程序入口，使用以下指令将 R14 存入堆栈：

```
STMFD  SP!,{<registers>,LR}
```

对应地，使用以下指令可完成子程序返回：

```
LDMFD  SP!,{<registers>,PC}
```

- 当发生异常时，该模式下对应的 R14 被设置为异常返回地址(有些异常有一个小的固定偏移量，在 2.4.3 中介绍)。

在其他情况下 R14 可作为通用寄存器使用。

(3) 程序计数器 R15

寄存器 R15 被用作程序计数器，也称为 PC。由于 ARM 处理器采用流水线机制，当正确读取 PC 时，该值为当前指令地址值加 8 字节。也就是说对于 ARM 指令来说，PC 指向当前指令的下两条指令的地址。在 ARM 状态下，PC 的第 0 位和第 1 位总是为 0；在 Thumb 状态下，PC 值的第 0 位总是为 0。当成功向 PC 写入一个地址数值时，程序将跳转到该地址执行。

PC 虽然也可作为通用寄存器，但一般不这样使用。因为对于 R15 的使用有一些特殊限制，违反了这些限制，程序执行结果未知。

2. 程序状态寄存器

CPSR(当前程序状态寄存器)可以在任何处理器模式下被访问，它包含了条件标志位、中断使能位、当前处理器模式标志以及其他的一些控制和状态位。

每一种异常模式下又都有一个专用的物理状态寄存器，称为程序状态备份寄存器(SPSR)。当特定的异常发生时，SPSR 用于保存 CPSR 的当前值，在异常中断程序退出时，可以用 SPSR 中保存的值来恢复 CPSR。

CPSR 和 SPSR 格式相同，它们的格式如图 2.10 所示。

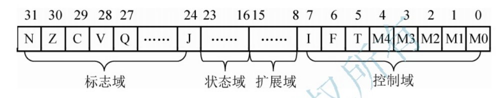


图 2.10 程序状态寄存器的格式

(1) 标志域

➢ 条件标志位。N(Negative)、Z(Zero)、C(Carry)、V(Overflow)均为条件标志位。它们的值可被算数或逻辑运算的结果所改变，并且大部分的 ARM 指令可以根据 CPSR 中的这些条件标志位来选择是否执行。各条件标志位的具体含义如表 2-5 所示。

表 2-5 条件标志位的具体含义

标志位	含义
N	符号标志位。本位设置成当前指令运算结果的 bit[31]的值。 当两个补码表示的有符号整数运算时，N=I 表示运算的结果为负数；N=0 表示结果为正数或零
Z	结果为 0 标志位。Z=1 表示运算的结果为零；Z=0 表示运算的结果不为零。 对于 CMP 指令，Z=1 表示进行比较的两个数大小相等
C	进位或借位标志位。下面分四种情况讨论 C 的设置方法。 在加法指令中(包括比较指令 CMN)，当结果产生了进位，则 C=1，表示无符号数运算溢出；其他情况下 C=0。 在减法指令中(包括比较指令 CMP)，当运算中发生借位，则 C=0，表示无符号数运算溢出；其他情况下 C=1。 对于包含移位操作的非加/减法运算指令，C 被置为移出值的最后 1 位。 对于其他非加/减法运算指令，C 位的值通常不受影响
V	溢出标志位。对于加/减法运算指令，当操作数和运算结果为二进制的补码表示的带符号数时，V=1 表示符号位溢出。 通常其他的指令不影响 V 位，具体可参考各指令的说明

➢ Q 标志位。在 ARM V5 的 E 系列处理器中，CPSR 的 bit[27]称为 Q 标志位，主要用于指示增强的 DSP 指令是否发生了溢出。

在 ARM V5 以前的版本及 ARM V5 的非 E 系列的处理器中，Q 标志位没有被定义。

- J 标志位。该标志位为 Jazelle 状态标志位，在 V5TEJ 架构及以后被定义。J=1 表示处理器处于 Jazelle 状态。

(2) 控制域

CPSR 的低 8 位(包括 I、F、T 及 M[4:0])称为控制位，当发生异常时这些位发生变化。如果处理器运行特权模式，这些位也可以由程序修改。

- 运行模式控制位 M[4:0]。控制位 M[4:0]控制处理器模式，具体含义如表 2-6 所示

表 2-6　运行模式控制位 M[4:0]的具体含义

M[4:0]	模式	可见的 ARM 状态寄存器
10000	用户模式	R0～R14，PC，CPSR
10001	FIQ 模式	R0～R7，R8_fiq～R14_fiq，PC，CPSR，SPSR_fiq
10010	IRQ 模式	R0～R12，R13_irq，R14_irq，PC，CPSR，SPSR_irq
10011	管理模式	R0～R12，R13_svc，R14_svc，PC，CPSR，SPSR_svc
10111	数据访问中止模式	R0～R12，R13_abt，R14_abt，PC，CPSR，SPSR_abt
11011	未定义指令中止模式	R0～R12，R13_und，R14_und，PC，CPSR，SPSR_und
11111	系统模式	R0～R14，PC，CPSR

注：处理器模式既可以通过程序直接改写 CPSR(特权模式下)来切换，也可以当内核响应异常时由硬件切换。

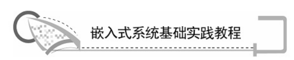
- 中断禁止位 I、F。当 I=1 时，禁止 IRQ 中断；当 F=1 时，禁止 FIQ 中断。
- T 控制位。指令执行状态控制位，用来说明本指令是 ARM 指令还是 Thumb 指令。对于 ARM V4 以及更高版本的 T 系列的 ARM 处理器：T=0 表示执行 ARM 指令；T=1 表示执行 Thumb 指令。

对于 ARM V5 以及更高版本的非 T 系列的 ARM 处理器：T=0 表示执行 ARM 指令；T=1 表示强制下一条执行的指令产生未定义指令中断。

CPSR 中的其他位用于 ARM 版本的扩展。

2.4　异　　常

异常通常的定义是处理器中止正常的程序执行流程并转向相应的处理。例如，处理一个外设的中断请求或者读写数据时发生存储器故障都会引起异常处理。在处理异常之前，当前的状态应该保留，这样当异常处理完成后，当前的程序可以继续执行。处理器允许多个异常同时发生，它们会按照固定的优先级顺序进行处理。在本节中将对 ARM 的异常机制作简要介绍。



2.4.1 ARM 体系结构支持的异常类型

在 ARM 体系结构中，存在七种异常处理，它们分别是复位、未定义指令、软件中断(SWI)、指令预取中止、数据中止、外部中断请求(IRQ)和快速中断请求(FIQ)。当异常发生时 CPU 自动到指定的向量地址读取指令并且执行，即 ARM 的向量地址处存放的是一条指令(一般是一条跳转指令)，跳转到专门处理某个异常的子程序。

当多个异常同时发生时，处理器会按照固定的优先级顺序来处理异常。表 2-7 列出了 ARM 的七种异常及它们的优先级顺序。

表 2-7 ARM 的七种异常

异常类型	处理器模式	优先级	向量表偏移
复位	管理模式	1	0x00000000
未定义指令	未定义指令中止模式	6	0x00000004
软件中断 SWI	管理模式	6	0x00000008
指令预取中止	数据访问中止模式	5	0x0000000c
数据中止	数据访问中止模式	2	0x00000010
保留	/	/	0x00000014
外部中断请求(IRQ)	外部中断模式	4	0x00000018
快速中断请求(FIQ)	快速中断模式	3	0x0000001c

1. 复位

复位具有最高的优先级。当处理器的复位引脚有效时，系统产生复位异常。转到复位异常处理程序处执行，复位异常通常用在两种情况：系统加电时；系统复位时。

复位异常处理程序将进行一些初始化工作，如实现设置异常中断向量表、初始化所有模式下的堆栈及寄存器、初始化存储系统及初始化关键的 I/O 设备等功能。

2. 未定义指令

当 ARM 处理器遇到一条不属于 ARM 或 Thumb 指令集的指令时，ARM 会“询问”协处理器，看它能否将其当成一条协处理器指令来处理。如果这条指令不属于任何一个协处理器，则会产生未定义指令异常。

在没有物理协处理器(硬件)的系统上，在未定义指令异常处理程序中可对协处理器进行软件仿真，或者在软件仿真时进行指令扩展。

3. 软件中断 SWI

软件中断 SWI 异常发生时，处理器进入管理模式，用于用户模式下的程序调用特权操作指令。在实时操作系统中可以通过该机制实现系统功能调用。

4. 指令预取中止

如果处理器预取的指令的地址不存在，或者该地址不允许当前指令访问，存储器系统向 ARM 处理器发出存储器中止(Abort)信号，预取的指令被记为无效。但只有当处理器试



图执行无效指令时，指令预取中止异常才会发生，如果指令未被执行，例如，在指令流水线中发生了跳转，则预取指令中止异常不会发生。

5. 数据中止

如果存储器访问指令(Load/Store)的目标地址不存在，或者该地址不允许当前指令访问，处理器产生数据访问中止异常中断。

6. 外部中断请求(IRQ)

当处理器的 IRQ 引脚有效，而且 CPSR 寄存器的 I 控制位被清除时，处理器产生外部中断异常。系统中各外部设备通常通过该异常中断请求处理器服务。

7. 快速中断请求(FIQ)

FIQ 异常是为了支持数据传输或者通道处理而设计的。当处理器的 FIQ 引脚有效，而且 CPSR 寄存器的 F 控制位被清除时，处理器产生 FIQ。

以下两个特性能够让处理器尽快地响应 FIQ。

1) 如表 2-7 所示，FIQ 中断向量在异常向量表的最后，这样使 FIQ 处理程序可以直接从 FIQ 向量处开始，省去了跳转的时间开销。

2) FIQ 模式下有五个私有寄存器(R8_fiq～R12_fiq)，对于这些寄存器在进入和退出 FIQ 时无须保存和恢复，节省了时间。

2.4.2　异常的响应

ARM 处理器响应异常的过程如下。

1) 将下一条指令的地址存入相应的异常模式的链接寄存器 LR，以便程序在处理异常返回时能从正确位置重新开始执行。

2) 复制 CPSR 寄存器的内容至对应模式下的 SPSR_<mode>寄存器中。

3) 对 CPSR 寄存器的一些控制位进行设置：

无论发生异常时处理器处于 Thumb 状态还是 ARM 状态，响应异常后处理器都会切换到 ARM 状态，即 CPSR[5]=0；

将模式控制位 CPSR[4:0]设置为被响应异常的模式编码；

CPSR[7]=1，禁止 IRQ 中断；

如果异常模式为复位模式或 FIQ 模式，则 CPSR[6]=1，禁止 FIQ 中断。

4) 将程序计数器(PC)设置为异常向量的地址，使程序从相应的异常向量地址开始执行异常处理程序。一般来说，向量地址处将包含一条指向相应异常处理程序的转移指令，从而可以跳转到相应异常处理程序处。

注：以上操作都是 ARM 处理器核硬件逻辑自动完成。

2.4.3　异常的返回

复位异常无需返回，因为系统复位后将开始整个用户程序的执行。其他异常在返回时，异常处理程序应实现下列操作：

- 将 SPSR_<mode>中的内容恢复到 CPSR 中；
- 将 LR 的值减去偏移量后送入 PC，偏移量根据异常类型不同而有所区别，如表 2-8 所示；
- 若在进入异常处理时设置了中断禁止位，要在此清除。

1. 异常返回指令

具体的异常返回指令如表 2-8 所示。

表 2-8　异常和返回地址

异常类型	返回指令	含义
软件中断 SWI	MOVS　PC，R14_svc	指向 SWI 指令的下一条指令
未定义指令	MOVS　PC，R14_und	指向未定义指令的下一条指令
指令预取中止	SUBS　PC，R14_abt，#4	指向导致预取指令中止异常的那条指令
快速中断请求	SUBS　PC，R14_fiq，#4	FIQ 处理程序的返回地址
外部中断请求	SUBS　PC，R14_irq，#4	IRQ 处理程序的返回地址
数据中止	SUBS　PC，R14_abt，#8	指向导致数据中止异常的那条指令
复位	无	没有定义 LR

表 2-8 中，MOV 指令和 SUB 指令尾部有一个“S”，并且 PC 是目的寄存器，表示 CPSR 将自动从 SPSR 中恢复。

返回的另外一种方法：如果 LR 寄存器已经被修正，而且异常处理程序已经把返回地址保存到堆栈，也可以使用堆栈操作指令来恢复用户寄存器并实现返回，举例如下：

```
SUB      LR,LR,#8              ;修正返回地址
STMFD    SP!,{R0-R4, LR}       ;保存使用到的寄存器及返回地址到堆栈
…                              ;异常处理代码
LDMFD    SP!,{R0-R4, PC}^      ;中断返回
```

其中，中断返回指令的寄存器列表(必须包含 PC)后的“^”符号表示这条指令在装载 PC 的同时，CPSR 的值也从 SPSR 中得到恢复。这里使用的堆栈指针 SP(R13)属于异常模式的寄存器，每个异常模式有自己的堆栈指针。

2. 返回地址的修正

在异常响应时，处理器会对 LR 做一次自动调整：ARM 状态下设置 LR_<mode>=PC−4，其中 PC 指向当前正在取指的指令；而在 Thumb 状态下，LR_<mode>会被自动修正。

自动调整完成后，可以根据具体的异常类型进一步修正返回地址，如下所述。

(1) 从 SWI 和未定义指令返回

SWI 和未定义指令异常中断是由当前执行的指令自身产生的，当 SWI 和未定义指令异常中断产生时，程序计数器 PC 的值还未更新，它指向当前正在执行指令后面第 2 条指令(对于 ARM 指令来说+8 字节，对于 Thumb 指令来说+4 字节)。这时处理器自动修正后即指向返回后将要执行的指令，无需再修正。



```
                    ARM         Thumb
            SWI     PC-8        PC-4      ;异常发生处
指令 1   ▲          PC-4        PC-2      ;LR=将要执行指令地址
指令 2              PC          PC
```

返回操作可以通过下面的指令来实现：

```
MOVS  PC, LR
```

▲ 表示异常返回后将要执行的指令。

(2) 从 FIQ、IRQ 异常返回

FIQ 与 IRQ 异常中断一样，在处理器执行完当前指令后，查询 FIQ 及 IRQ 中断引脚，如果中断引脚有效，并且系统允许该中断产生，处理器将产生 FIQ 或 IRQ 异常中断。此时，PC 的值已经更新，它指向当前指令后面第三条指令(对于 ARM 指令，它指向当前指令地址+12 字节；对于 Thumb 指令，它指向当前指令地址+6 字节)。这时在处理器自动修正后再减 4 字节即为返回后将要执行的指令的地址。

```
               ARM       Thumb
指令 1         PC-12     PC-6       ;指令执行结束后产生异常
指令 2   ▲     PC-8      pc-4
指令 3         PC-4      PC-2       ;ARM:LR=下一条指令地址
指令 4         PC        PC         ;Thumb:LR=下两条指令地址
```

返回操作可以通过下面的指令来实现：

```
SUBS  PC, LR, #4
```

▲ 表示异常返回后将要执行的指令。

(3) 从指令预取中止异常返回

当发生指令预取中止异常中断时，程序要返回到该有问题的指令处，重新读取并执行该指令。这种异常是由当前执行的指令自身产生的，程序计数器 PC 的值还未更新，它指向当前指令后面第二条指令。这时在处理器自动修正后再减 4 字节即为返回后将要执行的指令的地址。

```
               ARM       Thumb
指令 1   ▲     PC-8      PC-4       ;异常发生在此指令执行期间
指令 2         PC-4      PC-2       ;ARM:LR=下一条指令地址
指令 3         PC        PC         ;Thumb:LR=下两条指令地址
```

返回操作可以通过下面的指令来实现：

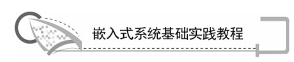
```
SUBS  PC, LR, #4
```

▲ 表示异常返回后将要执行的指令。

(4) 从数据中止异常返回

发生数据访问异常中断时，程序要返回到该有问题的指令处，重新访问该数据。数据





访问中止异常中断发生时，程序计数器 PC 的值已经更新，它指向当前指令后面第 3 条指令(对于 ARM 指令，它指向当前指令地址+12 字节；对于 Thumb 指令，它指向当前指令地址+6 字节)。这时需在处理器自动修正后再减 8 字节即为返回后将要执行的指令的地址。

```
                 ARM       Thumb
指令 1    ▲    pC-12     pC-6      ;异常发生处
指令 2         pC - 8    pC-4
指令 3         pC-4      pC-2      ;ARM:LR=下两条指令地址
指令 4         pc        pc
指令 5         pc+4      pc+2      ;Thumb:LR=下四条指令地址
```

返回操作可以通过下面的指令来实现：

```
SUBS  PC,LR,#8
```

▲ 表示异常返回后将要执行的指令。

2.5 存储方式及存储器管理单元

2.5.1 大、小端格式

ARM 处理器将存储器看做是一个从 0 开始的线性递增的字节集合。字节 0～3 保存第 1 个存储的字，字节 4～7 保存第 2 个存储的字，字节 8～11 保存第三个存储的字，依此排列。作为 32 位的微处理器，ARM 体系结构所支持的最大寻址空间为 4GB。

ARM 处理器对存储器操作的数据类型包括字节(8 位)、半字(16 位)和字(32 位)。

在 ARM 中，要求地址 A 是字对齐的，有下面几种：

- 地址为 A 的字单元包括字节单元 A，A+1，A+2，A+3；
- 地址为 A 的半字单元包括字节单元 A，A+1；
- 地址为 A+2 的半字单元包括字节单元 A+2，A+3；
- 地址为 A 的字单元包括半字单元 A，A+2。

ARM 体系结构可以用两种方式存储数据，称之为小端格式(Little-Endian)和大端格式(Big-Endian)。

1. 小端格式

在小端存储格式中，低地址中存放的是字数据或半字数据的低字节，而高地址中存放的是字数据或半字数据的高字节，如图 2.11 所示。

<table>
<tr><td>31　　24</td><td>23　　16</td><td>15　　8</td><td>7　　0</td></tr>
<tr><td colspan="4">字单元A</td></tr>
<tr><td colspan="2">半字单元A+2</td><td colspan="2">半字单元A</td></tr>
<tr><td>字节单元A+3</td><td>字节单元A+2</td><td>字节单元A+1</td><td>字节单元A</td></tr>
</table>

图 2.11　以小端格式存储数据



2. 大端格式

在大端存储格式中，与小端存储格式相反，低地址中存放的是字数据或半字数据的高字节，而高地址中存放的是字数据或半字数据的低字节，如图 2.12 所示。

31 24	23 16	15 8	7 0
字单元A			
半字单元A		半字单元A+2	
字节单元A	字节单元A+1	字节单元A+2	字节单元A+3

图 2.12 以大端格式存储数据

【例 2-1】存储一个 32 位数 0x2356873 到 3000H～3003H 四个字节单元中，分别以大端格式和小端格式存储，试分析 3000H 存储单元的内容。

解：采用小端格式存储数据如图 2.13(a)所示，(3000H)=73H；采用大端格式存储数据如图 2.13(b)所示，(3000H)=02H。

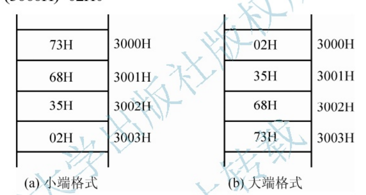


图 2.13 大、小端格式存储示意图

一个基于 ARM 内核的芯片可以只支持大端格式或小端格式，也可以两者都支持。通常，小端格式是 ARM 处理器的默认形式。

在 ARM 指令集中不包含任何直接选择大小端的指令，但是一个同时支持大小端格式的 ARM 芯片可以通过硬件配置(一般使用芯片的引脚来配置)来匹配存储器系统所使用的规则。

2.5.2 存储器管理单元

针对各种 CPU，存储器管理单元(Memory Management Unit，MMU)是个可选的配件。MMU 提供的一个关键的服务是使各个任务作为各自独立的程序在其自己的私有存储空间中运行。在带有 MMU 的操作系统控制下，运行的任务无须知道其他与之无关的任务的存储需求情况，这就简化了各个任务的设计。

操作系统通过使用处理器的 MMU 功能，可以实现虚拟内存，可以在处理器上运行比实际物理内存大的应用程序。MMU 作为转换器，将程序和数据的虚拟地址转换成实际的物理地址，即在物理主存中的地址。这个转换过程允许运行的多个程序使用相同的虚拟地址，而各自存储在物理存储器的不同位置。

在 ARM 处理器系列中，ARM7 系列一般不含有 MMU 单元(ARM720T 除外)；ARM9

系列一般是含有 MMU 单元的，但 ARM940T 只有 MPU(Memory protection unit，存储保护单元)，不是一个完整的 MMU；ARM11 系列均含有 MMU 单元；在 ARM Cortex 系列中，Cortex-A 系列处理器均有 MMU 单元，而 Cortex-R 系列和 Cortex-M 系列一般具有可选 MPU 单元，无 MMU 单元；SecurCore 系列均支持 MPU，无 MMU 单元。

2.6 案 例 分 析

本章导入案例中介绍了 ARM 处理器在全球范围内的广泛应用，ARM 处理器的广泛应用与其设计思想密不可分，ARM 内核采用 RISC 体系结构。RISC 是一种设计思想，其目标是设计出一套能在高时钟频率下单周期执行，简单而有效的指令集。下面用本章所学知识结合 RISC 思想对 ARM 的设计思想作简单分析。

2.6.1 RISC 思想在 ARM 处理器设计中的体现

1. 流水线

指令的处理过程被拆分成几个更小的、能够被流水线并行执行的单元。在理想情况下，流水线每周期前进一步，可获得最高的吞吐率。基本上每一种 ARM 处理器内核都支持流水线机制，如 ARM7 的三级流水线，ARM9 的五级流水线，而 Cortex-A9 处理器的设计基于最先进的推测型八级流水线，该流水线具有高效、动态长度、多发射超标量及无序完成等特征，使得时钟主频进一步提高。

2. 寄存器

RISC 处理器拥有大量的寄存器，每个寄存器都可存放数据或地址。ARM 处理器具有 31 个通用寄存器，有些是各模式共用的同一个物理寄存器，有些是各模式自己独有的物理寄存器。

3. 指令集

RISC 处理器减少了指令种类。RISC 的指令种类只提供简单的操作，使一个周期就可以执行一条指令。编译器或者程序员通过几条简单指令的组合来实现一个复杂的操作。每条指令的长度都是固定的，允许流水线在当前指令译码阶段去取下一条指令。在 ARM 指令集中，每条指令都为 32 位。

4. Load/Store 结构

处理器只处理寄存器中的数据。独立的 Load/Store 指令用来完成数据在寄存器和外部存储器之间的传送。ARM 处理器的指令中包含单寄存器传送的 Load/Store 指令、多寄存器传送的 LDM/STM 指令及交换指令完成存储器访问操作，这将在下一章中做详细介绍。

2.6.2 ARM 设计思想中的改进之处

1. 一些特定指令的周期数可变

并不是所有的 ARM 指令都是单周期。例如，多寄存器 Load/Store 指令的执行周期就



是不确定的，须根据被传送的寄存器个数而定，ARM 多寄存器指令允许一条指令最多传送 16 个寄存器。

2. 内嵌桶形移位器产生了更为复杂的指令

内嵌桶形移位器是一个硬件部件，在一个输入寄存器被一条指令使用之前，内嵌桶形移位器可以处理该寄存器中的数据。它扩展了许多指令的功能，以此改善了内核性能，提高了代码密度。

3. Thumb 16 位指令集

ARM 内核增加了一套称之为 Thumb 指令的 16 位指令集，使得内核既能够执行 16 位指令，也能够执行 32 位指令，从而增强了 ARM 内核的性能。16 位指令与 32 位的定长指令相比较，代码密度可提高约 30%。

4. 条件执行

只有当某个特定条件满足时指令才会被执行。这个特性可以减少分支指令的数目，从而改善性能，提高代码密度。ARM 指令集中几乎所有的指令都支持条件执行，下一章中会作介绍。

5. 增强指令

一些功能强大的数字信号处理器(DSP)指令被加入到标准的 ARM 指令之中，以支持快速的 16×16 位乘法操作及饱和运算。

这些增加的特性使得 ARM 处理器成为当今最通用的 32 位嵌入式处理器内核之一。

本 章 小 结

目前 32 位的嵌入式微处理器以 ARM 为核心，本章介绍了 ARM 处理器指令集体系的发展历史，以及各个版本的典型处理器及应用情况和性能分析；分析了 ARM 处理器的内核结构；讲解了 ARM 处理器的编程模型及异常机制；最后介绍了 ARM 处理器的存储格式。

(1) ARM CPU 的指令集体系 ISA 从最初的 V1 版本发展到现在，先后出现了 V1，V2，V3，V4，V5，V6，V7，V8 等版本。每一种指令集体系版本可以由多种处理器实现。目前市场上流通使用的 ARM 微处理器主要包括 ARM7、ARM9、ARM11 和 ARM Cortex 系列，以及专门为安全设备设计的 SecurCore 系列。

(2) 典型 ARM 处理器的内核结构：介绍了 ARM7TDMI、ARM9TDMI 及 ARM Cortex-A9 三种处理器内核的结构。ARM7 内核采用冯 • 诺依曼体系结构、9MIPS 的三级流水线；而 ARM9 内核采用哈佛体系结构，流水线也从三级增加到 1.1MIPS/MHZ 的五级；而 Cortex-A9 处理器的设计基于最先进的推测型八级流水线。

(3) ARM 的编程模型：介绍了 ARM 处理器的 ARM 和 Thumb 两种工作状态；分析了 ARM 的七种运行模式；介绍了 ARM 处理器的寄存器组织，包括 31 个通用寄存器及六个

状态寄存器。

(4) 异常：介绍了 ARM 体系结构的异常机制。ARM 处理器支持七种异常类型，介绍了每种异常的产生、异常响应的过程及从异常返回时处理器执行的操作和异常返回指令。

(5) 存储格式及 MMU：主要介绍了 ARM 处理器存储数据的两种格式，大端格式和小端格式；简要叙述了 MMU 的功能。



阅读材料

1. CISC 和 RISC 指令集

常见的 CPU 指令集分为 CISC 和 RISC 两种。

CISC(Complex Instruction Set Computer)是“复杂指令集”。自 PC 诞生以来，32 位以前的处理器都采用 CISC 指令集方式。由于这种指令系统的指令不等长，因此指令的数目非常多，编程和设计处理器时都较为麻烦。但由于基于 CISC 指令架构系统设计的软件已经非常普遍了，所以包括 Intel、AMD 等众多厂商至今使用的仍为 CISC。

RISC(Reduced Instruction Set Computing)是“精简指令集”。研究人员在对 CISC 指令集进行测试时发现，各种指令的使用频度相当悬殊，其中最常使用的是一些比较简单的指令，它们仅占指令总数的 20%，但在程序中出现的频度却占 80%。RISC 正是基于这种思想提出的。采用 RISC 指令集的微处理器处理能力强，并且还通过采用超标量和超流水线结构，大大增强并行处理能力。

2. NEON 多媒体技术

NEON 技术在 ARM V7 内核版本引入，可加速多媒体和信号处理算法(如视频编码/解码、2D/3D 图形、游戏、音频和语音处理、图像处理技术、电话和声音合成)，其性能至少为 ARM V5 性能的三倍，为 ARM V6 SIMD 性能的两倍。

NEON 技术具有以下优点：支持用于 Internet 应用程序范围广泛的多媒体编解码器；NEON 可使复杂视频编解码器的性能提高 60%～150%，单个简单 DSP 算法可实现更大的性能提升(4～8 倍)，处理器可更快进入睡眠状态，从而在整体上节约了动态功耗；NEON 技术的大量元素能够提高性能并简化软件开发过程。

3. TrustZone 安全技术

ARM TrustZone 技术是系统范围的安全方法，在 ARM V6 内核版本引入，针对高性能计算平台上的大量应用，包括安全支付、数字版权管理(DRM)和基于 Web 的服务。

TrustZone 安全技术通过以下方式确保系统安全：隔离所有 SoC 硬件和软件资源，使它们分别位于两个区域(用于安全子系统的安全区域以及用于存储其他所有内容的普通区域)中。硬件逻辑可确保普通区域组件无法访问任何安全区域资源，从而在这两个区域之间构建强大边界。将敏感资源放入安全区域的设计，以及在安全的处理器内核中可靠运行软件可确保资产能够抵御众多潜在攻击，包括那些通常难以防护的攻击(例如，使用键盘或触摸屏输入密码)。

支持 TrustZone 技术的 ARM 处理器包括 ARM Cortex-A15、ARM Cortex-A9、ARM Cortex-A8、ARM Cortex-A5 及 ARM1176。

4. Jazelle 技术

Jazelle 是 ARM 体系结构的一种相关技术，用于在处理器指令层次对 Java 加速。首颗具备 Jazelle 技术的 ARM 处理器是 ARM926EJ-S，包含 Jazelle 技术的处理器以一个英文字母“J”标示于 CPU 名称中。

ARM Jazelle 软件包括在任何现有 JVM 和 Java 平台中支持 Jazelle 硬件的技术。它还包括功能丰富的

多任务虚拟机(MVM)，领先的手机供应商和 Java 平台软件供应商提供的许多 Java 平台中均集成了此类虚拟机。通过利用基础 Jazelle 技术体系结构扩展，ARM MVM 软件解决方案可提供高性能应用程序和游戏，快速启动和应用程序切换，并且使用的内存和功耗预算非常低。

习　　题

一、选择题

1．存储一个 32 位数 0x2168465 到 2000H～2003H 四个字节单元中，若以大端格式存储，则 2000H 存储单元的内容为(　　)。

A．0x21　　B．0x68　　C．0x65　　D．0x02

2．寄存器 R13 除了可以做通用寄存器外，可还以做(　　)。

A．程序计数器　　B．链接寄存器
C．栈指针寄存器　　D．基址寄存器

3．ARM 公司专门从事(　　)。

A．基于 RISC 技术芯片设计开发　　B．ARM 芯片生产
C．软件设计　　D．ARM 芯片销售

二、判断题

1．ARM7TDMI 内核采用哈佛体系结构。　　(　　)

2．寄存器 R15 被用作程序计数器 PC。　　(　　)

3．当多个异常同时发生时，处理器会按照固定的优先级顺序来处理异常，指令预取中止异常的优先级高于数据中止异常。　　(　　)

三、问答题

1．简述 ARM 有几种运行模式？

2．ARM 有哪几种异常类型？

3．ARM 处理器数据的存储格式有哪两种？有什么区别？

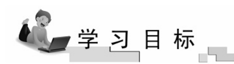
4．ARM7TDMI 采用几级流水线？每一级完成什么样的功能？

5．CPSR 中哪些位用来定义处理器状态？

6．各种异常的返回指令是什么？

第 3 章

ARM 嵌入式微处理器指令集



学习目标

理解 ARM 处理器指令集的特点及条件执行的含义；
熟悉 ARM 指令集的寻址方式、指令、伪指令及伪操作；
掌握 ARM 汇编语言程序的设计方法。

知识结构

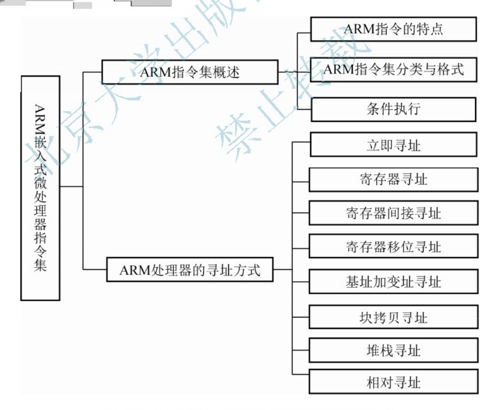


图 3.1　ARM 嵌入式微处理器指令集知识结构图



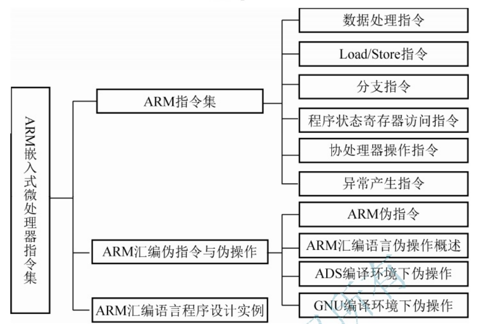


图 3.1　ARM 嵌入式微处理器指令集知识结构图(续)



导入案例

在基于 ARM 的嵌入式软件开发中，即使大部分程序使用高级语言完成，但与处理器硬件相关的部分还是必须使用汇编语言来编写。例如，ARM 启动代码使用汇编语言初始化处理器模式、设置堆栈、初始化变量等，这些操作均与处理器体系结构和硬件控制器相关。

本案例以在 S3C2410 上的启动代码为例介绍 ARM 汇编语言的应用。所谓启动代码，就是系统上电或复位后进入 C 语言的函数 main( )前执行的一段代码，如我们熟知的 BIOS 引导程序的功能。ARM 启动代码从系统上电开始接管 CPU，依次需要负责设置中断向量表、系统寄存器配置、CPU 在各种模式下的堆栈空间、设定 CPU 的内存映射、对 CPU 的外部存储器进行初始化、设定各外围设备的基地址、为 C 代码执行创建 ZI 区，然后进入到 C 代码。启动代码的一般流程如图 3.2 所示。

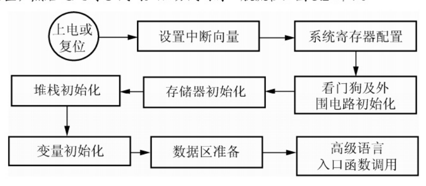


图 3.2　启动代码流程

前面以 μC/OS-II 在 S3C2410 上的启动代码为例体现了 ARM 汇编语言在嵌入式软件开发中的必要性。本章将着眼于 ARM 指令集，讲解 ARM 指令集的语法格式、寻址方式，分类介绍 ARM 指令、伪指令及伪操作的详细功能以及在使用中的注意事项，并给出 ARM 汇编语言编程的实例，为后续嵌入式软件开发奠定汇编语言基础。

3.1 ARM 指令集概述

3.1.1 ARM 指令的特点

ARM 处理器支持 ARM 指令集、Thumb 指令集和 Thumb-2 指令集。

从 ARM7TDMI 开始，ARM 处理器一直支持两种形式上相对独立的指令集，它们分别为 32 位的 ARM 指令集和 16 位的 Thumb 指令集。对应处理器状态分别为 ARM 状态和 Thumb 状态。

ARM 指令具有很高的执行效率。但是由于每条指令都要占用 4 个字节，对于存储空间的要求较高。为了压缩代码的存储，增加代码存储密度，ARM 公司设计了 16 位的 Thumb 指令集。

严格来讲，Thumb 不是一个完整的指令体系，Thumb 指令集实现的功能只是 32 位 ARM 指令集的子集，它仅仅把常用的 ARM 指令压缩成 16 位的指令编码方式，不能期望处理器只执行 Thumb 指令而不支持 ARM 指令集。因此，Thumb 指令只需要支持通用功能，必要时可以借助完善的 ARM 指令集。这时就要涉及处理器状态的切换。

从 ARM 体系结构 V6 版本开始，支持 Thumb-2 指令集。例如，ARM1156T2-S 内核便支持 Thumb 和 Thumb-2 两种指令集。Thumb-2 是一种新型混合指令集，融合了 16 位和 32 位指令，用于实现密度和性能的最佳平衡。在同时支持 16 位和 32 位指令之后，就无需在 Thumb 状态和 ARM 状态之间来回切换了。

本章重点介绍 32 位的 ARM 指令集。

3.1.2 ARM 指令集分类与格式

ARM 指令集总体可以分为六大类，具体指令功能见 3.3 节：

- 数据处理指令；
- Load/Store 指令；
- 分支指令；
- 程序状态寄存器访问指令；
- 协处理器操作指令；
- 异常产生指令。

ARM 指令字长为固定的 32 位，一条典型的 ARM 指令的格式为

```
    < opcode > { < cond > } {s}  < Rd >, < Rn >{, < operand2 >}
```

其中，<>内的项是必需的，{}内的项是可选的。例如，<opcode>是指令操作码，这是必须书写的；而{<cond>}为指令执行条件，是可选项，若不书写则无条件执行。

opcode　　指令操作码，如 ADD、MOV 等。

cond　　指令执行条件，如 EQ、NE 等。

S　　决定指令的操作结果是否影响 CPSR 寄存器的值。书写 S 时影响 CPSR。



Rd　　　目标寄存器。

Rn　　　包含第一个操作数的寄存器。

operand2　第二个操作数。采用的寻址方式可以为立即寻址、寄存器寻址及寄存器移位寻址。详见 3.2 节寻址方式部分。

3.1.3　条件执行

几乎所有的 ARM 指令均可以包含一个可选的条件码，而对于 Thumb 指令集，只有分支指令 B 具有条件码，语法说明中以{ <cond>}表示。只有在 CPSR 中的条件码标志满足指定的条件时，带条件码的指令才能执行；否则指令被忽略。使用指令条件码可实现高效的逻辑操作，提高代码效率。可以使用的指令条件码如表 3-1 所示。

表 3-1　指令条件码表

操作码	条件码助记符后缀	标志	含义
0000	EQ	Z=1	相等
0001	NE	Z=0	不相等
0010	CS/HS	C=1	无符号数大于或等于
0011	CC/LO	C=0	无符号数小于
0100	MI	N=1	负数
0101	PL	N=0	正数或零
0110	VS	V=1	溢出
0111	VC	V=0	没有溢出
1000	HI	C=1 且 Z=0	无符号数大于
1001	LS	C=0 且 Z=1	无符号数小于或等于
1010	GE	N=V	有符号数大于或等于
1011	LT	N!=V	有符号数小于
1100	GT	Z=0 且 N=V	有符号数大于
1101	LE	Z=1 且 N!=V	有符号数小于或等于
1110	AL	任何	无条件执行(默认)
1111	NV	任何	从不执行

【例 3-1】比较 R0 和 10 的大小，并进行相应的赋值处理。

```
CMP     R0,#10        ;R0 与 10 比较
MOVHI   R1,#1         ;若 R0 > 10,则 R1=1
MOVLS   R1,#0         ;若 R0 ≤ 10,则 R1=0
```

3.2　ARM 处理器的寻址方式

寻址方式是根据指令中给出的地址码字段来寻找真实操作数地址的方式。ARM 处理器支持的基本寻址方式有以下几种。

3.2.1 立即寻址

立即寻址是一种特殊的寻址方式。操作数本身在指令中直接给出，取出指令也就取出了操作数，这样的操作数被称为立即数。对应的寻址方式被称为立即寻址方式。例如，以下指令：

```
MOV  R1,#0x10      ;R1←0x10
```

其中，用#开始表示立即数，0x 表示十六进制数。

合法的立即数由一个 8 位的常数进行 32 位循环右移偶数位得到，其中循环右移的位数由一个 4 位二进制的两倍表示。如果立即数记作< immediate >，8 位常数记作 immed_8，4 位循环右移值记作 rotate_imm，则有< immediate >=immed_8 进行 32 位循环右移(2 * rotate_imm)，只有能够通过上面的构造方法得到的立即数才是合法的立即数。例如，0xFF，0xFF00 是合法的立即数；0xFF01，0x101 是不合法的立即数。

3.2.2 寄存器寻址

在寄存器寻址方式下，寄存器中的数值即为操作数。这种寻址方式是各类微处理器经常采用的一种寻址方式。例如，以下指令：

```
MOV   R1,R2              ;R1←R2
ADD   R0,R1,R2           ;R0←R1+R2
```

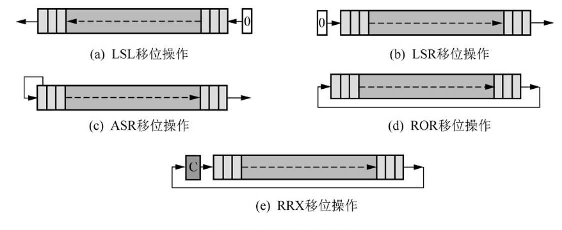
3.2.3 寄存器间接寻址

在寄存器间接寻址方式下，寄存器中的数值作为操作数的地址，而所需操作数是存放在该地址指定的存储单元中。例如，以下指令：

```
STR      R0,[R1]         ;将 R0 寄存器的值保存至 R1 指定的存储单元
LDR      R0,[R1]         ;将 R1 指定的存储单元的内容读出,保存至 R0 中
```

3.2.4 寄存器移位寻址

寄存器移位寻址方式是 ARM 指令集所特有的。指令中的第二操作数是寄存器的数值进行相应的移位而得到的，移位位数可以用立即数方式或者寄存器方式给出。可采用的移位操作有以下几种，如图 3.3 所示。



(a) LSL移位操作　(b) LSR移位操作

(c) ASR移位操作　(d) ROR移位操作

(e) RRX移位操作

图 3.3　移位操作过程



LSL：逻辑左移，空出的最低有效位用 0 填充。

LSR：逻辑右移，空出的最高有效位用 0 填充。

ASL：算术左移，由于左移空出的有效位用 0 填充，因此，它与 LSL 同义。

ASR：算术右移，对象是带符号数，移位过程中必须保持操作数的符号不变。如果源操作数是正数，空出的最高有效位用 0 填充，如果是负数用 1 填充。

ROR：循环右移，移出的最低有效位依次填入空出的最高有效位。

RRX：带扩展循环右移。将寄存器内容循环右移 1 位，空位用原来 C 标志位填充，移出的最低有效位填入 C 标志位。

寄存器移位方式举例如下：

```
MOV   R0,R1,LSL # 2          ;R0=R1 * 4
MOV   R0,R1,ASR # 2          ;R0=带符号数 R1/4
ADD   R0,R0,R1,LSL R3        ;R1 的值左移 R3 位,然后和 R0 相加,结果放入 R0
MOV   R0,R1,ROR R3           ;R0=R1 循环右移 R3 位
```

3.2.5　基址加变址寻址

基址加变址寻址就是将寄存器(该寄存器一般称作基址寄存器)的内容与指令中给出的地址偏移量相加，从而得到一个操作数的有效地址。这种寻址方式常用来访问某个基址附近的存储单元。

基址加变址的寻址方式又可以分成以下几种寻址方式。

1．前变址法

基址寄存器中的值和地址偏移量先作加减运算，生成的操作数作为内存访问的地址。举例如下：

```
LDR     R0,[ R1,# 4]         ;R0 ←[R1+4]
LDR     R0,[ R1,# 4] !       ;R0 ←[R1+4],R1=R1+4
```

在第一条指令中，将基址寄存器 R1 的内容加上偏移量 4 形成操作数的有效地址，然后从中取出操作数存入寄存器 R0 中。指令执行完毕后基址寄存器的内容不变。

在第二条指令中，将基址寄存器 R1 的内容加上偏移量 4 形成操作数的有效地址，从而取出操作数存入寄存器 R0 中；然后将 R1 的内容加 4。

注：“!”表示在完成数据传送后将更新基址寄存器。

2．后变址法

将基址寄存器中的值直接作为内存访问的地址进行操作，内存访问完毕后基址寄存器中的值和地址偏移量作加减运算，并更新基地址寄存器。举例如下：

```
LDR     R0,[ R1],# 4              ;R0 ←[R1],R1=R1 + 4
```

在这条指令中，基址寄存器 R1 的值作为操作数的有效地址，从中取出操作数存入寄存器 R0 中，然后将 R1 的内容加 4 来更新基址寄存器。

在上述两种变址方法的举例中，地址偏移量是用立即数表示，事实上它也可以是另一个寄存器或者是寄存器移位的方式。举例如下：

```
LDR     R0,[ R1,R2 ] !          ;R0 ←[R1+R2],R1=R1+R2
LDR     R0,[ R1],R2             ;R0 ←[R1],R1=R1+R2
LDR     R0,[ R1],R2,LSL #2      ;R0 ←[R1],R1=R1+R2 * 4
```

3.2.6　块拷贝寻址

块拷贝寻址也叫多寄存器寻址，可以在存储器中的数据块和寄存器组之间进行数据传递。允许一条指令最多完成传送 16 个寄存器的值。块拷贝寻址的地址变化方式有以下四类型。

1) 后增 IA(Increment After) ：每次数据传送后地址加 4。
2) 先增 IB (Increment Before) ：每次数据传送前地址加 4。
3) 后减 DA(Decrement After) ：每次数据传送后地址减 4。
4) 先减 DB (Decrement Before) ：每次数据传送前地址减 4。

块拷贝指令举例如下：

```
STMIA  R0，{R1-R3，R8}      ;[ R0 ]←R1
                            ;[ R0+4 ]←R2
                            ;[ R0+8 ]←R3
                            ;[ R0+12 ]←R8
LDMIA  R0，{R1-R3，R8}      ;R1←[ R0 ]
                            ;R2←[ R0+4 ]
                            ;R3←[ R0+8 ]
                            ;R8←[ R0+12 ]
```

使用块拷贝寻址方式的指令时，寄存器组的顺序由小到大排列，连续的寄存器可用“-”连接，或用“，”分隔。

3.2.7　堆栈寻址

堆栈是一种数据结构，按先进后出(First In Last Out，FILO)的方式工作，使用一个称做堆栈指针的专用寄存器指示当前的操作位置，堆栈指针总是指向栈顶，在 ARM 中常用 R13 作为栈指针(SP)。

根据堆栈指针的指向位置不同可将堆栈分为满堆栈和空堆栈。

- 满堆栈(Full Stack)：当堆栈指针指向最后压入堆栈的数据时；
- 空堆栈(Empty Stack)：当堆栈指针指向下一个将要放入数据的空位置时。

根据堆栈的生成方式，又可以分为递增堆栈和递减堆栈。

- 递增堆栈(Ascending Stack)：当堆栈由低地址向高地址生成时；



- 递减堆栈(Descending Stack)：当堆栈由高地址向低地址生成时。
  ARM 处理器支持以下四种类型的堆栈工作方式。
- 满递增堆栈 FA(Full Ascending)：堆栈指针指向最后压入的数据，且由低地址向高地址生成；
- 满递减堆栈 FD(Full Descending)：堆栈指针指向最后压入的数据，且由高地址向低地址生成；
- 空递增堆栈 EA(Empty Ascending)：堆栈指针指向下一个将要放入数据的空位置，且由低地址向高地址生成；
- 空递减堆栈 ED(Empty Descending)：堆栈指针指向下一个将要放入数据的空位置，且由高地址向低地址生成。

堆栈寻址指令举例如下：

```
   STMFD  SP!,{R1-R3,R8}     ；将 R1-R3、R8 共四个寄存器依次入栈
```

栈操作其实也是块拷贝操作，每一条栈操作指令都相应与一条块拷贝操作相对应，其对应关系如表 3-2 所示。

表 3-2　块拷贝与栈操作的对应关系

		递增		递减	
		满	空	满	空
增	先增	STMIB STMFA			LDMIB LDMED
	后增		STMIA STMEA	LDMIA LDMFD	
减	先减		LDMDB LDMEA	STMDB STMFD	
	后减	LDMDA LDMFA			STMDA STMED

3.2.8　相对寻址

相对寻址方式以程序计数器 PC 的当前值为基地址，指令中的地址标号作为偏移量，两者相加后的地址即为操作数的有效地址。举例如下：

```
           BL   SUB1                 ;跳转到子程序 SUB1 处执行
           …
    SUB1                             ;子程序入口
           …
           MOV   PC,LR               ;从子程序返回
```

以上程序段完成了子程序的调用和返回，跳转指令 BL 采用了相对寻址方式。

3.3 ARM 指令集

3.3.1 数据处理指令

ARM 的数据处理指令包括数据传送指令、算术运算指令、逻辑运算指令、比较指令和乘法指令，下面分别加以介绍。

1. 数据传送指令

数据传送指令是最简单也是最常用的 ARM 指令，常用于赋初值及寄存器间的数据传送。ARM 数据传送指令如表 3-3 所示。

表 3-3 数据传送指令

助记符	说明	操作
MOV{cond}{S} Rd，operand2	数据传送	Rd←operand2
MVN{cond}{S} Rd，operand2	数据非传送	Rd←(～operand2)

(1) 数据传送指令 MOV

MOV 指令的格式为

```
MOV{cond}{S}  Rd,operand2
```

MOV 指令可以将第二操作数 operand2 表示的数据传送到目标寄存器 Rd 中；其中 S 选项决定指令的操作是否影响 CPSR 中条件标志位的值，当没有 S 时，指令不更新 CPSR 中的条件标志位的值。

MOV 指令举例如下：

```
MOV  R0,R1              ;R0←R1
MOV  R1,#0x20           ;R1←0x20
MOV  R1,R2,LSL #2       ;R1←R2 * 4
```

注：当 PC 寄存器作为目标寄存器时可以实现程序跳转。这种跳转可以实现子程序调用以及从子程序中返回，如“MOV PC，LR”。

当 PC 寄存器作为目标寄存器且指令中包含 S 后缀时，指令在执行跳转操作的同时，将当前处理器模式的 SPSR 寄存器内容复制到 CPSR 中。这样可以实现从某些异常中断中返回，如“MOVS PC，LR”。但对于用户模式和系统模式，指令执行结果未知，因为这两种模式下无寄存器 SPSR。

(2) 数据非传送指令 MVN

MVN 指令的格式为

```
MVN{cond}{S}  Rd,operand2
```

MVN 指令可以将第二操作数 operand2 表示的数据进行按位逻辑“非”操作后传送到目标寄存器 Rd 中。



MVN 指令举例如下：

```
MVN  R1,#0               ;将立即数 0 按位取反后传送到 R1 中，完成后 R1=-1
```

2. 算术运算指令

算术运算指令用于实现 32 位有符号或无符号数的加法和减法运算，如表 3-4 所示。

表 3-4　算数运算指令

助记符	说明	操作
ADD{cond}{S}　Rd，Rn，operand2	加法运算指令	Rd←Rn＋operand2
SUB{cond}{S}　Rd，Rn，operand2	减法运算指令	Rd←Rn−operand2
RSB{cond}{S}　Rd，Rn，operand2	逆向减法指令	Rd←operand2−Rn
ADC{cond}{S}　Rd，Rn，operand2	带进位加法指令	Rd←Rn＋operand2+Carry
SBC{cond}{S}　Rd，Rn，operand2	带借位减法指令	Rd←Rn−operand2−(NOT)Carry
RSC{cond}{S}　Rd，Rn，operand2	带借位逆向减法指令	Rd←operand2−Rn−(NOT)Carry

其中，S 选项决定指令的操作是否影响 CPSR 中条件标志位的值，当没有 S 时，指令不更新 CPSR 中的条件标志位的值。

【例 3-2】算术运算指令举例。

```
ADD      R0,R1,R2                ;R0←R1+R2
ADDS     R1,R0,R0,LSL #1         ;R1←R0*3,并根据运算结果更新 CPSR
SUB      R1,R0,#0x20             ;R1←R0-0x20
SUBS     R1,R0,R2,LSL #2         ;R1←R0-R2*4,并根据运算结果更新 CPSR
RSB      R1,R0,#0x20             ;R1←0x20-R0
RSBS     R1,R0,R2,LSL #2         ;R1←R2*4-R0,并根据运算结果更新 CPSR
```

【例 3-3】ADC 指令可以实现高于 32 位数的加法运算。

本例实现将 R0 和 R1 中的 64 位整数(R0 中存放低 32 位)与 R2 和 R3 中的 64 位整数(R2 中存放低 32 位)相加，结果存放在 R4 和 R5 中(R4 中存放低 32 位)。

```
ADDS     R4,R0,R2                ;低 32 位相加并影响标志位
ADC      R5,R1,R3                ;高 32 位相加再加上进位标志位
```

【例 3-4】SBC 指令可以实现高于 32 位数据的减法运算。

本例实现将 R0 和 R1 中的 64 位整数(R0 中存放低 32 位)与 R2 和 R3 中的 64 位整数(R2 中存放低 32 位)相减，结果放在 R4 和 R5 中(R4 中存放低 32 位)

```
SUBS     R4,R0,R2                ;低 32 位相减并影响标志位
SBC      R5,R1,R3                ;高 32 位相减再减去 C 标志位的反码
```

【例 3-5】使用 RSC 指令实现求 64 位数值的负数

```
RSBS    R2,R0,#0
RSC     R3,R1,#0                 ;使用 RSC 指令实现求 64 位数值的负数
```




注：当指令包含S选项时，如果减法运算有借位，则C=0，否则C=1。

3. 逻辑运算指令

逻辑运算指令可对两个操作数按位作逻辑操作，如表3-5所示。

表3-5　逻辑运算指令

助记符	说明	操作
AND{cond}{S}　Rd，Rn，operand2	逻辑“与”操作指令	Rd←Rn & operand2
ORR{cond}{S}　Rd，Rn，operand2	逻辑“或”操作指令	Rd←Rn | operand2
EOR{cond}{S}　Rd，Rn，operand2	逻辑“异或”操作指令	Rd←Rn^ operand2
BIC{cond}{S}　Rd，Rn，operand2	位清除指令	Rd←Rn &(~operand2)

其中，S选项决定指令的操作是否影响CPSR中条件标志位的值，当没有S时，指令不更新CPSR中的条件标志位的值。

(1) 逻辑“与”操作指令AND

AND指令的格式为

```
AND{cond}{S}  Rd,Rn,operand2
```

AND指令可用于提取寄存器中某些位的值而将其他位清0。将某一位与1作逻辑与操作，该位值将不变；将某位与0作逻辑与操作，该位值被清0。

AND指令举例如下

```
AND  R0,R0,#0xFF            ;将寄存器R0低8位保持不变,高24位清0
```

(2) 逻辑“或”操作指令ORR

ORR指令的格式为

```
ORR{cond}{S}  Rd,Rn,operand2
```

ORR指令可用于提取寄存器中某些位的值而将其他位置1。将某一位与1作逻辑或操作，该位值将被置1；将某位与0作逻辑或操作，该位值不变。

ORR指令举例如下：

```
ORR  R0,R0,#0xFF            ;将寄存器R0低8位置1,高24位保持不变
```

(3) 逻辑“异或”操作指令EOR

EOR指令的格式为

```
EOR{cond}{S}  Rd,Rn,operand2
```

EOR指令可用于将寄存器中某些位的值取反而其他位保持不变。将某一位与1作异或操作，该位值将被取反；将某位与0作异或操作，该位值不变。

EOR指令举例如下：

```
EOR  R0,R0,#0xFF            ;将寄存器R0低8位按位取反,高24位保持不变
```



(4) 位清除指令 BIC

BIC 指令的格式为

```
BIC{cond}{S}  Rd,Rn,operand2
```

BIC 指令将第二操作数 operand2 表示的数据的反码和寄存器 Rn 的值按位作逻辑“与”操作，并将结果保存至目标寄存器 Rd 中。

BIC 指令可用于提取寄存器中某些位的值而将其他位清 0。将寄存器的某一位与 1 作 BIC 操作，该位值将被清 0；将某位与 0 作 BIC 操作，该位值不变。

BIC 指令举例如下：

```
BIC  R0,R0,#0xFF              ;将寄存器 R0 低 8 位清 0,高 24 位保持不变
```

注：BIC 在清除标志位时是非常有用的，也经常用来清除 CPSR 中的中断控制位。

4. 比较指令

比较指令的共同特点是不保存运算结果，只用作更新 CPSR 中的条件标志位，指令操作码无需加 S 后缀。后面的指令可以根据相应的条件标志位来判断是否执行。ARM 比较指令如表 3-6 所示。

表 3-6　比较指令

助记符	说明	操作
CMP{cond}{S}　Rn，operand2	比较指令	标志 N、Z、C、V←Rn−operand2
CMN{cond}{S}　Rn，operand2	负数比较指令	标志 N、Z、C、V←Rn+operand2
TST{cond}{S}　Rn，operand2	位测试指令	标志 N、Z、C、V←Rn & operand2
TEQ{cond}{S}　Rn，operand2	相等测试指令	标志 N、Z、C、V←Rn ^ operand2

(1) 比较指令 CMP

CMP 指令的格式为

```
CMP{cond}{S}  Rn,operand2
```

CMP 指令用寄存器 Rn 的值减去第二操作数 operand2 表示的数值，根据操作结果来更新 CPSR 中的条件标志位，但不保存减法结果。

CMP 指令举例如下：

```
CMP  R0,#10                    ;R0 与 10 相比较,并更新相关标志位
```

注：CMP 指令与 SUBS 指令的区别在于 CMP 指令不保存结果。在进行两个数据的大小关系比较时，常用 CMP 指令及相应的条件码来判断。

(2) 负数比较指令 CMN

CMN 指令的格式为

```
CMN{cond}{S}  Rn,operand2
```

CMN 指令用寄存器 Rn 的值加上第二操作数 operand2 表示的数值，根据操作结果来更新 CPSR 中的条件标志位，但不保存运算结果。

CMN 指令举例如下：

```
CMN  R0,#10              ;R0 的值与 10 相加,并更新相关标志位.这条指令可以判断
                         ;R0 的值是否为 10 的补码,若是,则 Z=1
```

注：CMN 指令与 ADDS 指令的区别在于 CMN 指令不保存结果。

(3) 位测试指令 TST

TST 指令的格式为

```
TST{cond}{S}  Rn，operand2
```

TST 指令用寄存器 Rn 的值与第二操作数 operand2 表示的数值按位作逻辑“与”操作，根据操作结果更新 CPSR 中的条件标志位，但不保存运算结果。

TST 指令举例如下：

```
TST  R0,#0x01            ;判断 R0 寄存器的最低位是否为 0,如果是,则 Z=1
TST  R0,#0x0F            ;判断 R0 寄存器的低 4 位是否为 0,如果是,则 Z=1
```

注：TST 指令与 ANDS 指令的区别在于 TST 指令不保存相“与”的结果。在判断寄存器中某些位是否为 0 时，常用 TST 指令及相应的条件码来判断。

(4) 相等测试指令 TEQ

TEQ 指令的格式为

```
TEQ{cond}{S}  Rn,operand2
```

TEQ 指令用寄存器 Rn 的值与第二操作数 operand2 表示的数值按位作逻辑“异或”操作，根据操作结果更新 CPSR 中的条件标志位，但不保存运算结果。

TEQ 指令举例如下：

```
TEQ  R0,R1               ;判断 R0 的值与 R1 的值是否相等(不影响 V 位和 C 位)
```

注：TEQ 指令与 EORS 指令的区别在于 TEQ 指令不保存两个操作数“异或”的结果。TEQ 指令可用来判断两个操作数的值是否相等，若相等，则 Z=1；CPSR 中的 N 位为两个操作数的符号位作异或操作的结果。

5. 乘法指令

ARM 乘法指令完成两个寄存器中数据的乘法，按照保存结果的数据长度可以分两类：一类为 32 位的乘法指令，即乘法操作的结果为 32 位；另一类为 64 位的乘法指令，即乘法操作的结果为 64 位，如表 3-7 所示。

表 3-7　ARM 乘法指令

助记符	说明	操作
MUL{cond}{S}　Rd，Rm，Rs	32 位乘法	Rd←Rm * Rs　(Rd≠Rm)
MLA{cond}{S}　Rd，Rm，Rs，Rn	32 位乘加	Rd←Rm * Rs + Rn　(Rd≠Rm)
UMULL{cond}{S}　RdLo，RdHi，Rm，Rs	64 位无符号数乘法	RdHi：RdLo←Rm * Rs



续表

助记符	说明	操作
UMLAL{cond}{S}　RdLo，RdHi，Rm，Rs	64 位无符号数乘加	RdHi：RdLo=Rm * Rs +(RdHi：RdLo)
SMULL{cond}{S}　RdLo，RdHi，Rm，Rs	64 位有符号数乘法	RdHi：RdLo←Rm * Rs
SMLAL{cond}{S}　RdLo，RdHi，Rm，Rs	64 位无符号数乘加	RdHi：RdLo=Rm * Rs +(RdHi：RdLo)

说明：

RdLo、RdHi 寄存器：ARM 结果寄存器。R15 不能用作 RdLo，RdHi，Rm 或 Rs，否则指令执行的结果不可预测。

在 MUL 和 MLA 指令中，Rd 和 Rm 不能为同一个寄存器；在 UMULL、UMLAL、SMULL 和 SMLAL 指令中，RdLo，RdHi 和 Rm 不能为同一个寄存器，否则指令执行的结果不可预测。

S 后缀决定指令的操作是否影响 CPSR 中的 N 位和 Z 位，当有 S 后缀时指令更新 CPSR 中条件标志位。

乘法指令举例如下：

```
MUL     R0,R1,R2          ;R0←R1 * R2
MLAS    R0,R1,R2,R3       ;R0←R1 * R2 + R3,同时设置 CPSR 中相关条件标志位
UMULL   R0,R1,R2,R3       ;(R1:R0)←R2 * R3
UMLAL   R0,R1,R2,R3       ;(R1:R0)←R2 * R3 +(R1:R0)
SMULL   R0,R1,R2,R3       ;(R1:R0)←R2 * R3
SMLAL   R0,R1,R2,R3       ;(R1:R0)←R2 * R3 +(R1:R0)
```

注：两个 32 位数相乘的结果为 64 位，但由于 MUL 和 MLA 指令只保存了 64 位结果的低 32 位，所以不管操作数为有符号数或无符号数，MUL 和 MLA 指令的结果相同。

6. 其他数据处理指令

从 ARMV5 版本指令系统开始支持前导零计数指令 CLZ，其指令的格式为

```
CLZ{<cond>} Rd，Rm
```

其中，Rd 不允许是 R15。这条指令主要用于计算 32 位寄存器 Rm 操作数从第 31 位开始连续“0”的个数，直到遇到“1”停止计数并将“0”的个数送回目标寄存器 Rd。

3.3.2　Load/Store 指令

Load/Store 内存访问指令用于在寄存器和存储器之间传送数据，ARM 处理器对存储器的访问只能使用 Load/Store 指令。ARM 指令集中有三种基本的 Load/Store 内存访问指令。

- 单寄存器传送指令：LDR/STR 指令可以把单一的数据传入或传出一个寄存器，支持字(32 位)、半字(16 位)和字节操作；
- 多寄存器传送指令：LDM/STM 指令可实现一条指令 Load/Store 多个寄存器的内容；
- 交换指令：该指令可实现把一个存储单元的内容和寄存器内容相交换。

1．单寄存器传送 LDR 和 STR 指令

单寄存器传送指令如表 3-8 所示。

表 3-8　LDR/STR 指令

助记符	说明	操作
LDR{cond}　Rd，addressing	加载字数据	Rd←[addressing]
LDR{cond}B　Rd，addressing	加载无符号字节数据	Rd←[addressing]
LDR{cond}T　Rd，addressing	以用户模式加载字数据	Rd←[addressing]
LDR{cond}BT　Rd，addressing	以用户模式加载无符号字节数据	Rd←[addressing]
LDR{cond}H　Rd，addressing	加载无符号半字数据	Rd←[addressing]
LDR{cond}SB　Rd，addressing	加载有符号字节数据	Rd←[addressing]
LDR{cond}SH　Rd，addressing	加载有符号半字数据	Rd←[addressing]
STR{cond}　Rd，addressing	存储字数据	[addressing]←Rd
STR{cond}B　Rd，addressing	存储字节数据	[addressing]←Rd
STR{cond}T　Rd，addressing	以用户模式存储字数据	[addressing]←Rd
STR{cond}BT　Rd，addressing	以用户模式存储字节数据	[addressing]←Rd
STR{cond}H　Rd，addressing	存储半字数据	[addressing]←Rd

(1) 字和无符号字节访问指令

LDR 指令从内存中将一个 32 位的字数据或一个 8 位的字节数据读取至寄存器中，STR 指令将一个 32 位的寄存器中的字数据或一个寄存器的低 8 位写入到指令指定的内存单元中。

指令的格式为

```
LDR{cond}{T}   Rd,addressing      ;加载指定存储单元中的 32 位字数据到寄存器 Rd
STR{cond}{T}   Rd,addressing      ;存储 Rd 中的字数据到指定的存储单元中
LDR{cond}B{T}  Rd,addressing      ;加载字节数据至 Rd 的低 8 位，高 24 位清零
STR{cond}B{T}  Rd,addressing      ;将 Rd 的低 8 位写入指定的字节存储单元中
```

说明：

1) addressing 为访问的内存单元地址构成形式，它可采用的寻址方式有寄存器间接寻址和基址加变址寻址，详见 3.2 节 ARM 指令集的寻址方式。

2) T 为可选后缀，若指令有 T，则表示即使处理器是在特权模式下，存储系统也将访问看成是在用户模式下。T 在用户模式下无效，T 不能与前变址形式一起使用。

3) 当 LDR 指令的目标寄存器是 PC 时，可实现程序跳转功能。

加载/存储字和无符号字节指令举例如下：

```
LDR     R0,[R1]              ;将存储单元地址为 R1 中的字数据读入寄存器 R0 中
LDR     R1,[R0,#4]           ;将存储单元地址为 R0+4 中的字数据读入寄存器 R1 中
LDRB    R1,[R0,R2]           ;将存储单元地址为 R0+R2 中的字节数据读入寄存器 R1
                             ;的低 8 位,高 24 位清零
LDR     R1,[R0,R2,LSL #2]    ;将存储单元地址为 R0+4*R2 中的字数据读入寄存器 R1
```



```
中
        STR     R0,[R1]            ;将 R0 中的字数据写入以 R1 为地址的字存储单元中
        STRB    R1,[R0,#4]         ;将 R1 的低 8 位写入以 R0+4 为地址的字节存储单元中
```

(2) 半字和有符号字节访问指令

这类 LDR/STR 指令可实现加载有符号字节数据，加载有符号半字数据，Load/Store 无符号半字数据。

其指令的格式为

```
LDR{cond}SB   Rd,addressing   ;加载有符号字节数据至 Rd 低 8 位,高 24 位符号扩展
LDR{cond}SH   Rd,addressing   ;加载有符号半字数据至 Rd 低 16 位,高 16 位符号扩展
LDR{cond}H    Rd,addressing   ;加载无符号半字数据至 Rd 低 16 位,高 16 位清零
STR{cond}H    Rd,addressing   ;将 Rd 的低 16 位写入指定的内存单元
```

说明：

1) addressing 为访问的内存单元地址构成形式，它可采用的寻址方式有寄存器间接寻址和基址加变址寻址，详见 3.2 节 ARM 指令集的寻址方式。

2) 地址对齐：对半字传送的内存地址必须为半字对齐，否则指令会产生不可预知的后果。

3) 当 LDR 指令的目标寄存器是 PC 时，可实现程序跳转功能。

Load/Store 字和无符号字节指令举例如下：

```
LDRSB  R0,[R1]              ;将存储单元地址为 R1 中的有符号字节数据读入寄存器 R0
                            ;中,R0 的高 24 位用符号位扩展
LDRSH   R1,[R0,#4]          ;将存储单元地址为 R0+4 中的有符号半字数据读入寄存器
                            ;R1 中,R1 的高 16 位用符号位扩展
LDRH  R1,[R0,R2]            ;将存储单元地址为 R0+R2 中的无符号半字数据读入寄存器
                            ;R1 中,R1 的高 16 位清零
LDRH  R1,[R0,R2,LSL #2]     ;将存储单元地址为 R0+4*R2 中的无符号半字数据读入寄
                            ;存器 R1 中,R1 的高 16 位清零
STRH  R0,[R1]               ;将 R0 的低 16 位写入以 R1 为地址的存储单元中
STRH  R1,[R0,#4]            ;将 R1 的低 16 位写入以 R0+4 为地址的存储单元中
```

2. 多寄存器传送 LDM 和 STM 指令

多寄存器 Load/Store 指令可实现在一组寄存器和连续的内存单元之间的数据传输。LDM 指令将连续内存单元数据加载到若干个寄存器中，而 STM 指令将若干个寄存器的值存放到连续的内存单元中。允许一条指令传送 16 个寄存器(R0～R15)的任意子集(或全部)。LDM/STM 指令的主要用途有现场保护、数据复制和参数传递等。

LDM 和 STM 指令功能如表 3-9 所示。

表 3-9　LDM/STM 指令

助记符	说明	操作
LDM{cond}{mode}　Rn{!}，reglist{^}	多寄存器加载	reglist←[Rn…]，{Rn 回写}
STM{cond}{mode}　Rn{!}，reglist{^}	多寄存器存储	[Rn…]←reglist，{Rn 回写}



说明：

mode 有八种，前面四种为块拷贝操作，后面四种是堆栈操作，如下所示。

- IA：每次数据传送后地址加 4;
- IB：每次数据传送前地址加 4;
- DA：每次数据传送后地址减 4;
- DB：每次数据传送前地址减 4;
- FA：满递增堆栈;
- FD：满递减堆栈;
- EA：空递增堆栈;
- ED：空递减堆栈。

Rn 为基址寄存器，装有传送数据的初始地址，Rn 不允许为 R15。

“!”为可选后缀，若有“!”，则基址寄存器的值随着传送过程递增或递减；如果没有“!”，则基址寄存器的值保持不变。

Reglist 为 Load/Store 的寄存器列表，可包含多于一个寄存器或寄存器范围，使用“,”隔开，如{R1，R2，R6-R9}，寄存器排列由小到大排列。

“^”后缀不能在用户模式或系统模式下使用。该后缀表示当指令为 LDM 且寄存器列表中包含 PC 时，选用该后缀表示除了正常数据传送外，还将 SPSR 复制到 CPSR，可用于异常处理返回；如果不包括 R15，选用该后缀还表示传入传出的是用户模式下的寄存器，而不是当前模式下的寄存器。

注：指令中寄存器和连续内存单元的对应关系，即编号低的寄存器对应内存低地址单元，编号高的寄存器对应内存高地址单元。

LDM 和 STM 指令举例如下：

```
STMFD   R13! ,{R1-R3,R8}        ;将 R1-R3、R8 共 4 个寄存器依次入栈
LDMFD   R13! ,{R1-R3,R8}        ;是上面指令的出栈指令
STMIA   R0! ,{R5-R7}            ;将 R5-R7 的数据存储到 R0 指向的存储区域，R0 更新
LDMIA   R1! ,{R5-R7}            ;加载 R1 指向的存储区域的多字数据至 R5-R7 中
```

多寄存器传送指令示意图如图 3.4 所示，图中表明了 R5-R7 三个寄存器如何存到存储器中，以及基址寄存器的值如何自动修改。其中 R0 为指令执行前的基址寄存器，R0'为指令执行后的基址寄存器。

3．交换指令

交换指令能在一条指令内完成寄存器和存储器之间的数据交换。使用交换指令可实现信号量操作。交换指令在执行期间不能被其他任何指令或其他任何总线访问打断，在此期间系统占据总线，直至交换完成。交换指令功能如表 3-10 所示。

表 3-10　交换指令

助记符	说明	操作
SWP{cond}　Rd，Rm，[Rn]	寄存器和存储器字数据交换	Rd← [Rn]，[Rn]←Rm(Rn≠ Rd 或 Rm)
SWP{cond}B　Rd，Rm，[Rn]	寄存器和存储器字节数据交换	Rd 低 8 位←[Rn]，[Rn]←Rm 低 8 位(Rn≠Rd 或 Rm)



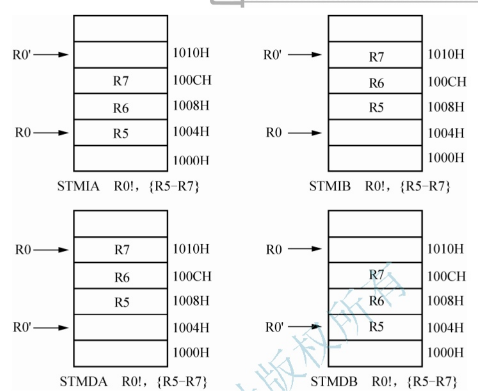


图 3.4　多寄存器传送指令示意图

其中，Rn 和 Rd 不能为同一个寄存器，Rn 和 Rm 也不能为同一个寄存器；当 Rd 和 Rm 为同一个寄存器时，SWP 指令将寄存器和内存单元的内容互换，SWPB 指令将寄存器低 8 位和内存字节单元内容互换。

交换指令举例如下：

```
SWP     R0,R2,[R3]    ;将 R3 指向的存储单元中的字数据传送至 R0 中,同时将 R2 中的
                      ;值存放到 R3 指向的存储单元中
SWP     R2,R2,[R3]    ;将 R3 指向的存储单元中的字数据和 R2 中的字数据互换
SWPB    R0,R2,[R3]    ;将 R3 指向的存储单元中的字节数据传送至 R0 的低 8 位,R0 高
                      ;24 位清 0;同时将 R2 的低 8 位存放到 R3 指向的字节存储单元中
SWPB    R2,R2,[R3]    ;将 R3 指向的存储单元中的字节数据和 R2 中的低 8 位数据互换
```

3.3.3　分支指令

ARM 分支指令可以改变指令执行的顺序。在 ARM 程序中有两种方式可以实现指令的跳转：一种是使用专门的分支指令；一种是直接向程序计数器 PC 写入跳转地址值。

直接向 PC 写入跳转地址值，可以实现在 4GB 地址空间中任意跳转，如果在跳转之前使用“MOV LR”等指令，可以保存程序的返回地址值，也就实现了在 4GB 地址空间中的子程序调用。

ARM 分支指令包括分支指令 B、带链接的分支指令 BL、带返回和状态切换的分支指令 BLX 以及带状态切换的分支指令 BX。ARM 分支指令如表 3-11 所示。

表 3-11　ARM 分支指令

助记符	说明	操作
B{cond}　target_address	分支指令	PC←target_address
BL{cond}　target_address	带链接的分支指令	PC←target_address，LR←PC
BLX　target_address 或 BLX {cond}　Rm	带链接和状态切换的分支指令	PC←target_address，LR←PC，切换处理器状态
BX{cond}　Rm	带状态切换的分支指令	PC←target_address，切换处理器状态

(1) 分支指令 B

B 指令使程序跳转到指定的地址执行程序。B 指令的格式为

```
B{cond}    target_address
```

target_address 存储在分支指令中的实际值是相对当前 PC 值的一个偏移量，而不是一个绝对地址。它的值由汇编器计算，它是 24 位有符号数，左移两位后有符号扩展为 32 位，表示的有效偏移为 26 位，也就是说 B 指令跳转范围为当前指令±32MB 地址空间。

B 指令和目标地址处的指令都要属于 ARM 指令集；B 指令加上{cond}后可以根据 CPSR 中的条件标志位决定指令是否执行。

B 指令应用举例如下：

```
1)  B        LABLE                ;程序无条件转移至标号 LABLE 处执行
    MOV      R0,#0x10
    …
LABLE
    SUB      R2,R1,# 4
    …
2)  CMP      R0,#0
    BNE      LABLE1               ;当 R0≠0 时,程序跳转至标号 LABLE1 处执行
```

(2) 带链接的分支指令 BL

带链接的分支指令 BL 将程序计数器 PC 的值保存至 R14(LR)中，然后跳转到指定的地址执行程序。

BL 指令的格式为

```
BL{cond}   target_address
```

target_address 的计算方法及 BL 指令的跳转范围同 B 指令。

BL 指令和目标地址处的指令都要属于 ARM 指令集；BL 指令加上{cond}后可以根据 CPSR 中的条件标志位决定指令是否执行。

BL 指令用于实现子程序调用。子程序的返回可以通过将 LR 寄存器的值复制到 PC 寄存器来实现。

【例 3-6】 使用 BL 指令跳转到一个子程序，再通过复制 LR 来返回。



```
            BL      func                ;调用 func 子程序
            CMP     R0,#10              ;比较 R0 和 10 的大小
            MOVLO   R1,#1               ;若 R0<10,则 R1=1
            …
    func
            …                           ;子程序代码
            MOV     PC,LR               ;返回
```

(3) 带状态切换的分支指令 BX

BX 指令使程序跳转到指令指定的地址执行程序，目标地址处的指令既可以是 ARM 指令，也可以是 Thumb 指令。

BX 指令的格式为

```
    BX{cond}    Rm
```

BX 指令跳转到 Rm 指定的地址执行，并根据 Rm 中的 bit[0]切换处理器的状态。若 Rm 中的 bit[0]为 1，则跳转时自动将 CPSR 中的 T 控制位置位，即把目标地址的代码解释为 Thumb 代码；若 Rm 中的 bit[0]为 0，则跳转时自动将 CPSR 中的 T 控制位清零，即把目标地址的代码解释为 ARM 代码。

注：当 Rm[1:0]=0b10 时，由于 ARM 指令是字对齐的，指令执行的结果会不可预知。

BX 指令应用举例如下：

```
    BX    R0        ;程序无条件跳转 R0 指定的地址,并根据 R0 的 bit[0]切换处理器状态
```


(4) 带链接和状态切换的分支指令 BLX

BLX 指令从 ARM 指令集跳转到指令中所指定的目标地址，并完成处理器工作状态的切换，该指令同时将程序计数器 PC 的当前内容保存到链接寄存器 R14 中。

根据目标地址的形式不同，BLX 指令有两种格式。

1) 由程序标号给出目标地址，指令的格式为

```
    BLX   target_address
```

无条件执行指令，目标地址始终为 Thumb 指令。该指令跳转范围为当前指令±32MB 地址空间。

2) 寄存器的内容作为目标地址，指令的格式为

```
    BLX{cond}  Rm
```

有条件执行指令，跳转到 Rm 指定的地址执行，并根据 Rm 中的 bit[0]来切换处理器的状态，切换方法同 BX 指令。

BLX 指令举例如下：

```
    BLX     R0          ;程序无条件跳转 R0 指定的地址,并根据 R0 的 bit[0]切换处理器状态
    BLX     thumbsub    ;调用 Thumb 代码子程序
```

当子程序使用 Thumb 指令集，而调用者使用 ARM 指令集时，可以通过 BLX 指令实现子程序调用和处理器状态切换。返回时要使用“BX　R14”指令来完成处理器状态切换。



3.3.4 程序状态寄存器访问指令

ARM 指令集提供了两条指令可以直接控制程序状态寄存器 PSR：MRS 指令用于将状态寄存器 CPSR 或 SPSR 的值读出到通用寄存器中；MSR 用于将一个寄存器的内容或一个立即数传送到 CPSR 或 SPSR 中。程序状态寄存器访问指令如表 3-12 所示。

表 3-12 程序状态寄存器访问指令

助记符	说明	操作
MRS{cond} Rd，PSR	读状态寄存器	Rd←PSR，PSR 为 CPSR 或 SPSR
MSR{cond} PSR_fields，Rm/immed_8r	写状态寄存器	PSR_fields←Rm/ immed_8r，PSR 为 CPSR 或 SPSR

1．读程序状态寄存器指令 MRS

MRS 指令用于将状态寄存器的内容传送到通用寄存器中。这是程序获得程序状态寄存器 PSR 值的唯一方法。

MRS 指令的格式为

```
MRS{cond} Rd,PSR
```

其中，Rd 不允许为 R15。PSR 为 CPSR 或 SPSR。

通过 MRS 指令可以取得程序状态寄存器的当前值，可以比较相应标志位了解当前 CPU 的状态及工作模式。

MRS 指令举例如下：

```
MRS      R0,CPSR        ;读取 CPSR 状态寄存器的值,保存到 R0 中
MRS      R1,SPSR        ;读取 SPSR 状态寄存器的值,保存到 R1 中
```

2．写程序状态寄存器指令 MSR

在 ARM 处理器中，只有 MSR 指令可以直接设置状态寄存器 CPSR 或 SPSR。MSR 可以将一个寄存器的内容或一个立即数传送到 CPSR 或 SPSR 中。

MSR 指令的格式为

```
MSR{cond} PSR_fields,Rm
MSR{cond} PSR_fields,immed_8r
```

其中，PSR 为 CPSR 或 SPSR。

fields　　状态寄存器中需要设置的区域。fields 可以是以下一种或多种：

c 控制域 PSR[7..0]

x 扩展域 PSR[15..8]

s 状态域 PSR[23..16]

f 标志域 PSR[31..24]

immed_8r 要传送到状态寄存器指定域的立即数，8 位



Rm　　　要传送到状态寄存器指定域的源寄存器

MSR 指令举例如下：

```
MSR      CPSR_c,#0xD3              ;CPSR[7:0]= 0xD3
MSR      CPSR_cxsf,R1              ;CPSR=R1
```

注：区域名必须为小写字母。向对应区域执行写入时，使用 PSR_fields 可以指定写入区域，而不影响状态寄存器其他位。

只有在特权模式下才能修改状态寄存器。

不能通过 MSR 指令直接修改 CPSR 中 T 位实现 ARM 状态和 Thumb 状态切换，必须使用 BX 指令来完成处理器状态切换。MRS 和 MSR 配合使用，可以实现对 CPSR 或 SPSR 寄存器的读—修改—写操作，可以切换处理器模式，或者允许/禁止 IRQ/FIQ 中断等。

【例 3-7】编写汇编语言程序段实现如下功能。

1) 切换处理器模式到 IRQ 模式。

```
MRS      R0,CPSR                   ;R0←CPSR
BIC      R0,R0,#0x1F               ;R0 低 5 位清零
ORR      R0,R0,#0x12               ;设置为 IRQ 模式
MSR      CPSR_c,R0                 ;传送回 CPSR
```

2) 禁止 IRQ 中断。

```
MRS      R0,CPSR                   ;R0←CPSR
ORR      R0,R0,#0x80               ;禁止 IRQ 中断
MSR      CPSR_c,R0                 ;传送回 CPSR
```


3.3.5　协处理器操作指令

ARM 的协处理器指令主要用于 ARM 处理器初始化 ARM 协处理器的数据处理操作；在 ARM 处理器的寄存器和协处理器的寄存器之间传送数据；以及在 ARM 协处理器的寄存器和存储器之间传送数据。这里对协处理操作指令作简要说明。ARM 协处理器操作指令如表 3-13 所示。

表 3-13　ARM 协处理器操作指令

助记符	说明	操作
CDP{cond}　coproc，opcode1，CRd，CRn，CRm{，opcode2}	协处理器数据操作指令	取决于协处理器
LDC{cond}{L}　coproc，CRd，<addr>	协处理器数据读取指令	取决于协处理器
STC{cond}{L}　coproc，CRd，<addr>	协处理器数据写入指令	取决于协处理器
MCR{cond}　coproc，opcode1，Rd，CRn，CRm{，opcode2}	ARM 寄存器到协处理器寄存器的数据传送指令	取决于协处理器
MRC{cond}　coproc，opcode1，Rd，CRn，CRm{，opcode2}	协处理器寄存器到 ARM 寄存器的数据传送指令	取决于协处理器

说明：

coproc	指令操作的协处理器名，标准名为 pn，n 为 0～15
opcode1	协处理器执行操作的第一操作码
opcode2	可选的协处理器的第二操作码
CRd	作为目标寄存器的协处理器寄存器
CRn	存放第一操作数的协处理器寄存器
CRm	存放第二操作数的协处理器寄存器
Rd	作为目的寄存器的 ARM 寄存器

协处理器指令举例如下：

```
CDP     p3,2,C8,C9,C5,6        ;协处理器 p3 初始化
LDC     p3,C2,[R0]             ;将 ARM 处理器的 R0 所指向的存储单元中数据传送到
                               ;协处理器 p3 的 C2 寄存器中
MCR     p3,2,R0,C1,C2,6        ;ARM 寄存器 R0 中数据传送到协处理器 p3 的寄存器中
```

注：协处理器操作指令只用于带协处理器的 ARM 核。

3.3.6　异常产生指令

ARM 指令集提供了两条异常产生指令，可以用软件的方法实现异常。ARM 异常产生指令如表 3-14 所示。

表 3-14　ARM 异常产生指令

助记符	说明	操作
SWI{cond}　immed_24	软中断指令	产生软件中断，处理器进入管理模式
BKPT　immed_16	断点中断指令	处理器产生软件断点

(1) 软件中断指令 SWI

SWI 指令用于产生软件中断，从而实现从用户模式变换到管理模式，CPSR 保存到管理模式的 SPSR 中，执行转移到 SWI 中断向量地址 0x08。ARM 通过这种机制使得用户程序可以调用操作系统的系统程序。

SWI 指令的格式为

```
SWI{cond}   immed_24
```

其中，immed_24 为 24 位立即数，指定了操作系统的服务类型，即软中断号。

使用 SWI 指令时，通常使用以下两种方法进行参数传递，SWI 异常中断处理程序就可以提供相关的服务。

1) 24 位立即数指定了用户程序调用系统例程的类型，相关参数通过通用寄存器传递。举例如下：

```
MOV    R0,#34              ;设置子功能号为 34
SWI    12                  ;调用中断号为 12 的软中断
```



2) 当指令中的 24 位的立即数被忽略时，用户程序调用系统例程的类型由通用寄存器 R0 的内容决定，同时，其他参数通过其他通用寄存器传递。举例如下：

```
MOV     R0,#12              ;设置 12 号软中断
MOV     R1,#34              ;设置子功能号为 34
SWI     0
```

(2) 断点中断指令 BKPT

断点中断指令可引起处理器进入调试模式，它使处理器停止执行正常指令而进入相应的调试程序。仅用于 V5T 体系。

断点中断指令 BKPT 的格式为

```
BKPT   immed_16
```

其中，immed_16 为 16 位的立即数，该立即数被调试程序用来保存额外的断点信息。

BKPT 指令举例如下：

```
BKPT    0x3202
```


3.4 ARM 汇编伪指令与伪操作

在进行 ARM 汇编语言编程时，除了要求掌握 3.3 节介绍的 ARM 指令集，还必须掌握 ARM 汇编伪操作和伪指令。本章将介绍进行 ARM 汇编语言程序设计时常用的伪指令，由 ARM 公司推出的开发工具所支持的伪操作及 GNU ARM 开发工具所支持的伪操作。

3.4.1 ARM 伪指令

伪指令不在处理器运行期间由机器执行，只是在汇编时被合适的机器指令代替成 ARM 或 Thumb 指令，从而实现真正的指令操作。ARM 伪指令包括 LDR、ADRL、ADR 和 NOP。

1. 大范围地址读取伪指令 LDR

LDR 伪指令将一个 32 位的立即数或者一个地址值读取到寄存器中。LDR 伪指令的格式为

```
LDR{cond}    register,=expr
```

其中，expr 可以是一个 32 位常数，也可以是程序代码中的标号。

在汇编编译源程序时，LDR 伪指令被编译器替换成一条合适的指令。若加载的常数未超出 MOV 或 MVN 的范围，则使用 MOV 或 MVN 指令代替该 LDR 伪指令；否则汇编器将常量放入文字池，并使用一条程序相对偏移的 LDR 伪指令从文字池读出常量，这时需要在 LDR 伪指令附近使用 LTORG 伪操作声明文字池。

【例 3-8】利用 LDR 伪指令加载立即数和标号地址。

```
LDR R1,= 0xAABBCCDD        ;将立即数 0xAABBCCDD 放入 R1 中
LDR R0,= place             ;将标号 place 地址放入 R0 中
…
LTORG                      ;声明文字池
…
```

注：LDR 伪指令处的 PC 值到文字池中的目标数据所在地址之间的偏移量必须小于 4KB 大小。

2. 中等范围地址读取伪指令 ADRL

ADRL 伪指令将基于 PC 相对偏移的地址值或基于寄存器相对偏移的地址值读取到寄存器中。在汇编编译器编译源程序时，ADRL 伪指令被编译器替换成两条合适的指令(通常是 ADD 或 SUB)。若不能用两条指令实现，则产生错误，编译失败。ADRL 伪指令的格式为

```
ADRL{cond}  register, expr
```

其中，expr 为地址表达式。当地址值是非字对齐时，取值范围为-64～64KB 之间；当地址值是字对齐时，取值范围为-256～256KB 之间。当地址是 16 字节对齐时，取值范围更大。

【例 3-9】ADRL 伪指令使用示例。

```
start
    MOV   R0,#0x10
    ADRL  R0,start         ;汇编后本条语句被 SUB  R0,PC,#12 及
                           ;NOP 两条语句替代
```

3. 小范围地址读取伪指令 ADR

它将基于 PC 相对偏移的地址值或基于寄存器相对偏移的地址值读取到寄存器中。通常，编译器用一条 ADD 指令或 SUB 指令来实现该 ADR 伪指令的功能，若不能用一条指令实现，则产生错误，编译失败。ADR 伪指令的格式为

```
ADR{cond}   register, expr
```

其中，expr 为地址表达式。当地址是字节对齐时，取值范围为-255～+255KB；当地址是字对齐时，取值范围是-1020～1020KB。当地址是 16 字节对齐时，取值范围更大。

【例 3-10】ADR 伪指令使用示例。

```
start
    MOV   R0,#0x10
    ADRL  R0,start              ;汇编后本条语句被 SUB  R0,PC,#0x0c 所替代
```

4. 空操作伪指令 NOP

NOP 在汇编时将会被替代成 ARM 中的空操作指令，如指令“MOV R0，R0”等。NOP 伪指令的格式为

```
NOP
```



NOP 伪指令可用作延时操作。

3.4.2 ARM 汇编语言伪操作概述

伪操作(Directive)是 ARM 汇编语言程序中的一些特殊的指令助记符，其作用主要是为完成汇编程序做各种准备工作，而不是在计算机运行期间由处理器执行。

伪操作只是在汇编过程中起作用，一旦汇编结束，伪操作也就随之消失。

伪操作与编译程序有关，在不同的编译环境下有不同的编写形式和规则。目前常用的 ARM 汇编程序的编译环境有两种。

- ADS/SDT、RealView MDK 等 ARM 公司推出的开发工具。ADS 由 ARM 公司开发，使用了 CodeWarrior 公司的编译器。MDK 是 Keil 公司(现已被 ARM 收购)开发的 ARM 开发工具，是用来开发基于 ARM 核的系列微控制器的嵌入式应用程序的开发工具。
- 集成了 GNU 工具的 IDE 编译环境。GNU 是“GNU's Not UNIX”的递归缩写。1983 年 9 月 27 日由 Richard Stallman 公开发起 GNU 计划，它的目标是创建一套完全自由的操作系统。

GNU 格式 ARM 汇编语言程序主要是面对在 ARM 平台上移植嵌入式 Linux 操作系统，GNU 组织开发的基于 ARM 平台的编译工具有主要由 GNU 的汇编器 as，交叉汇编器 gcc 和连接器 ld 等组成。

两种编译环境(本章简称 ADS 编译环境和 GNU 编译环境)下的伪操作不同，3.4.3 节和 3.4.4 节分别针对这两种编译环境下常用的伪操作进行简要介绍。

3.4.3 ADS 编译环境下的伪操作

ARM 公司推出的开发工具所支持的汇编伪操作主要包括符号定义伪指令、数据定义伪指令、汇编控制伪操作及其他功能伪操作。

1. 符号定义伪操作

符号定义伪操作用于在 ARM 汇编程序中定义变量、为变量赋值和为寄存器定义别名等操作。ADS 下常用的符号定义伪操作如表 3-15 所示。

表 3-15 ADS 下常用的符号定义伪操作

操作符	语法格式	作用
GBLA	GBLA variable	声明一个全局的算术变量，并将其初始化成 0
GBLL	GBLL variable	声明一个全局的逻辑变量，并将其初始化成{FALSE}
GBLS	GBLS variable	声明一个全局的字符串变量，并将其初始化成空串
LCLA	LCLA variable	声明一个局部的算术变量，并将其初始化成 0
LCLL	LCLL variable	声明一个局部的逻辑变量，并将其初始化成{FALSE}
LCLS	LCLS variable	声明一个局部的串变量，并将其初始化成空串
SETA	variable SETA expr	给一个全局或局部算术变量赋值
SETL	variable SETL expr	给一个全局或局部逻辑变量赋值

续表

操作符	语法格式	作用
SETS	variable　SETS　expr	给一个全局或局部字符串变量赋值
RLIST	name RLIST ｛list of registers｝	为一个通用寄存器列表定义名称

【例 3-11】声明一个局部算术变量 S1 并将其赋值为 0xaa；声明一个全局逻辑变量 S2 并将其赋值为{TRUE}；声明一个全局的字符串变量 S3 并将其赋值为“Testing”。

```
LCLA    S1
GBLL    S2
GBLS    S3
S1  SETA    0xaa
S2  SETL    {TRUE}
S3  SETS    "Testing"
```

【例 3-12】将寄存器列表 R0-R6，R8、R9 的名称定义为 Regilist。

```
Regilist    RLIST    {R0-R6,R8,R9}
```

注：LCLA、LCLL、LCLS 定义的局部变量在其作用范围内变量名必须唯一。GBLA、GBLL、GBLS 定义的全局变量在整个程序范围内变量名必须唯一。SETA、SETL、SETS 对变量进行赋值前必须声明变量。

2. 数据定义伪操作

数据定义伪操作一般用于为特定的数据分配存储单元，同时可完成已分配的存储单元的初始化。常见的数据定义伪操作如表 3-16 所示。

表 3-16　常用数据定义伪操作

操作符	语法格式	作用
DCB	{label} DCB expr｛，expr｝…	分配一段字节内存单元，并用 expr 初始化
DCW	{label} DCW expr｛，expr｝…	分配一段半字内存单元(半字对齐)
DCWU	{label} DCWU expr｛，expr｝…	分配一段半字内存单元(不要求半字对齐)
DCD	{label} DCD expr｛，expr｝…	分配一段字内存单元(字对齐)
DCDU	{label} DCDU　expr｛，expr｝…	分配一段字内存单元(不要求字对齐)
DCQ	{label} DCQ　expr｛，expr｝…	分配一个或多个双字内存块(字对齐)
DCQU	{label} DCQU　expr｛，expr｝…	分配一个或多个双字内存块(不要求字对齐)
LTORG	LTORG	声明一个数据缓冲池(literal pool)
SPACE	｛label｝SPACE　expr	分配一块连续字节内存单元，并用 0 初始化。

【例 3-13】数据定义伪操作使用示例。

```
Data    DCB  0x01,0x02,0x03        ;分配一段字节内存单元,并用1,2,3初始化
Str     DCB "Hello World! "        ;分配一段字节内存单元,并用字符串
                                   ;"Hello World"初始化
```



```
        Data1    DCD  2,4,6                  ;分配一段字内存单元,并用2,4,6初始化
        Dataspace     SPACE   100            ;分配连续100字节的存储单元并初始化为0
```

注：DCB 也可用“=”代替；DCD 也可用“&”代替；SPACE 也可用“%”代替。

【例 3-14】LTORG 伪操作使用示例。

```
    LDR      R0,=0X12345678
    ADD      R1,R1,R0
    MOV      PC,LR
    LTORG                          ;声明文字池，此地址存储0x12345678
```

在使用 LDR 伪指令时，要在适当的位置加入 LTORG 声明数据缓冲池，这样就会把要加载的数据保存到缓存池中，再使用 ARM 加载指令读出，如果没有使用 LTORG 声明数据缓冲池，则汇编器会在程序末尾自动声明。

注：LTORG 在子程序返回或无条件转移指令后使用。汇编程序对文字池中数据字对齐。

3. 汇编控制伪操作

汇编器在对程序代码进行编译时，根据汇编控制伪操作的定义情况对程序进行编译，常用的有条件编译、重复汇编和宏定义，如表 3-17 所示。

表 3-17　汇编控制伪操作

操作符	语法格式	作用
IF ELSE ENDIF	IF　logical expression … {ELSE …} ENDIF	能够根据条件成立与否把一段源代码包括在汇编语言程序内或者将其排除在程序之外。其中，ELSE 及其后指令序列可以没有
WHILE WEND	WHILE　logical expression … WEND	能够根据条件重复汇编相同的一段源代码
MACRO MEND MEXIT	MACRO {$label}　macroname {$parameter{，$parameter}…} …　；宏代码 MEND	用 MACRO 和 MEND 定义的一段代码，称为宏定义体。通过宏名称来调用宏。 MEXIT 用于从宏中跳转出去

【例 3-15】条件编译。

```
        GBLA val1
    val1 SETA 6
    IF val1 < 5
        MOV R0,#0                  ;条件成立则R0=0
    ELSE
        MOV R1,#0                  ;条件不成立则R1=0
    ENDIF
```

【例 3-16】重复汇编，实现一段程序代码循环编译五次。

```
                GBLA Count
        Count   SETA 1
        WHILE   Count    <=5
        …
        Count   SETA Count+1
        WEND
```

【例 3-17】带参数的宏定义。

```
    MACRO                                           ;宏定义
    $HandlerLabel    HANDLER  $HandlerLabel
    $HandlerLabel
        sub sp,sp,#4
            stmfd    sp!,{r0}
            ldr      r0,=$HandleLabel
            …
    MEND                                            ;宏定义结束
```

4. 杂项伪操作

ARM 汇编中还有一些其他的伪操作，在汇编程序中会经常被使用，如表 3-18 所示。

表 3-18 杂项伪操作

操作符	语法格式	作用
CODE16	CODE16	告诉编译器将要处理的为 16 位的 Thumb 指令
CODE32	CODE32	告诉编译器将要处理的为 32 位的 ARM 指令
EQU	name EQU expr {，type}	为程序中的常量、标号等定义一个等效的字符名称。EQU 可用“*”代替
AREA	AREA sectionname{，attr}{，attr}…	定义一个代码段或数据段
ENTRY	ENTRY	声明程序的入口点
END	END	通知汇编程序已到达源程序的结尾
EXPORT/ GLOBAL	EXPORT {symbol} {[WEAK{，attr}]}	声明一个符号可以被其他文件引用
IMPORT	IMPORT symbol {[WEAK]}	告诉编译器当前的符号不是在本源文件中定义的，而是在其他源文件中定义的，在本源文件中可以引用该符号。不论本源文件是否实际引用该符号，该符号将被加入到本源文件的符号表中
EXTERN	EXTERN symbol [WEAK{，attr}]	告诉编译器当前的符号不是在本源文件中定义的，而是在其他源文件中定义的。如果本源文件没有实际引用该符号，该符号不会被加入到本源文件的符号表中
GET/ INCLUDE	GET filename	将一个源文件包含到当前源文件中，并将被包含的文件在其当前位置进行汇编处理



续表

操作符	语法格式	作用
INCBIN	INCBIN　filename	将一个目标文件或数据文件包含到当前源文件中，被包含的文件不进行汇编处理
ALIGN	ALIGN　{expr {，offset {，pad}}}	通过添加字节的方式，使当前位置满足一定的对齐方式

(1) AREA 伪操作

AREA 伪操作用于定义一个代码段或数据段。语法格式为

```
AREA  sectionname {,attr} {,attr} …
```

其中，sectionname 指定所定义段的段名。段名若以数字开头，则该段名需用“|”括起来，如|1_test|。

attr 指定代码段或数据段的属性。各属性之间用逗号隔开，属性说明如表 3-19 所示。

表 3-19　段属性列表

段属性	说明
ALING=expr	在默认时，ELF (可执行连接文件)的代码段和数据段是按字对齐的，expr 可以取 0～31 的数值，相应的对齐方式为 2^{expr} 次方。例如，表达式=3 时为 8 字节对齐。expr 不能为 0 或 1
ASSOC=section	指定与本段相关的 ELF 段。任何时候连接 section 段也必须包括 sectionname 段
CODE	指定该段为代码段，默认为 READONLY 属性
COMDEF	该属性定义一个通用的段，该段可以包含代码或数据。在各源文件中，同名的 COMDEF 段必须相同
COMMON	定义一个通用数据段，不包含任何的用户代码和数据。连接器将其初始化为 0。各源文件中同名 COMMON 段共享同一段内存单元，连接器为其分配最大尺寸内存
DATA	指定该段为数据段，默认为 READWRITE
NOALLOC	指定该段为虚段，并不为其在目标系统上分配内存
NOINIT	指定本数据段不被初始化或仅初始化为 0。该操作仅为 SPACE/DCB/DCD/DCDU/DCQ/DCQU/DCW/DCWU 伪操作保留了内存单元
READONLY	指定该段不可写，为程序代码段
READWRITE	指定可读可写段，数据段的默认属性

注：通常可以用 AREA 伪操作将程序分为多个 ELF 格式的段。段名称可以相同，这时这些同名的段被放在同一个 ELF 段中。一个大的程序可以包括多个代码段和数据段，一个汇编语言程序至少要包含一个段。

(2) ENTRY 伪操作

ENTRY 伪操作声明程序的入口点。语法格式为

```
ENTRY
```

在一个完整的汇编程序中至少要有一个 ENTRY(也可以有多个，当有多个 ENTRY 时，程序的真正入口点由链接器指定)，但在一个源文件里最多只能有一个 ENTRY(可以没有)。

(3) EXPORT 伪操作

EXPORT(或 GLOBAL)伪操作用于在程序中声明一个全局的标号，该标号可在其他文件中引用，标号区分大小写。GLOBAL 是 EXPORT 的同义词。语法格式为

```
    EXPORT {symbol} {[WEAK{, attr}]}
```

其中，symbol 是要导出的符号名称，它是区分大小写的。如果省略 symbol，则导出所有符号。

WEAK 用于声明其他的同名标号优先该标号被引用。

attr 符号属性。用于定义符号对其他文件的“可见性”，如表 3-20 所示。

表 3-20 attr 属性

attr 属性	含义
DYNAMIC	符号可以被其他文件引用，且可以在其他文件中被重新定义
HIDDEN	符号不能被其他组件引用
PROTECTED	符号可以被其他文件引用，但不可重新定义

(4) END 伪操作

END 伪操作通知汇编程序已达到源程序的末尾。语法格式如下：

```
    END
```

每一个汇编源文件必须以 END 结束。如果汇编文件通过伪操作 GET 被指定了一个“父文件”，当汇编器遇到 END 伪操作时将返回到“父文件”继续汇编。

【例 3-18】杂项伪操作使用示例。

```
NUM EQU   0x1f                          ;定义 NUM 的值为 0x1f
        IMPORT  SVCStack                ;SVCStack 标号在其他源文件中定义
EXPORT  Boot                            ;声明一个可全局引用标号 boot
        AREA init,CODE,READONLY         ;定义一个代码段 init,属性为只读
        ENTRY                           ;指定程序入口点
        CODE32                          ;通知编译器后面的指令为 32 位 ARM 指令
START
        MOV R0, #NUM                    ;R0=0x1f
        …
        END                             ;文件结束
```

3.4.4 GNU 编译环境下的伪操作

在 GNU 编译环境下的伪操作包括符号定义伪操作、数据定义伪操作、代码控制伪操作和预定义控制伪操作。

1. 符号定义伪操作

GNU 编译环境下的符号定义伪操作如表 3-21 所示。



表 3-21　符号定义伪操作

伪操作	语法格式	说明
. equ	. equ　symbol，expr	将 symbol 定义为 expr
. set	. set　symbol，expr	作用同. expr
. equiv	. equiv　symbol，expr	symbol 定义为 expr，若 symbol 已定义过则出错
. global	. global　symbol	将 symbol 定义为全局标号
. globl	. globl　symbol	作用同. global
. extern	. extern　symbol	声明 symbol 为一个外部变量

【例 3-19】符号定义伪操作使用示例。

```
. equ        aa, 0xff              @将 0xff 定义成符号 aa
. global     start                 @声明全局标号 start
. extern     main                  @外部变量 main 在其他文件中声明
```

2. 数据定义伪操作

GNU 编译环境下的数据定义伪操作如表 3-22 所示。

表 3-22　数据定义伪操作

伪操作	语法格式	说明
. byte	.byte　expr {，expr} …	分配一段字节内存单元，并用 expr 初始化
. hword/. short	. hword　expr {，expr} …	分配一段半字内存单元，并用 expr 初始化(16bit)
.word/.long /.int	. word　expr {，expr} …	分配一段字内存单元，并用 expr 初始化(32bit)
. quad	. quad　expr {，expr} …	定义一段双字内存空间
. octa	. octa　expr {，expr} …	定义一段四字内存空间
. ascii	. ascii　expr {，expr} …	定义字符串 expr(非零结束符)
. asciz /. string	. asciz　expr {，expr} …	定义字符串 expr(以 0 为结束符)
. float/. single	. float　expr {，expr} …	定义一个 32bit IEEE 单精度浮点数 expr
. double	. double expr {，expr} …	定义 64bit IEEE 双精度浮点数 expr
. fill	. fill　repeat {，size}{，value}	分配一段字节内存单元，用 size 长度 value 填充 repeat 次。size 默认为 1，value 默认为 0
. zero	. zero size	分配一段字节内存单元，并用 0 填充(size 个字节)
. space/. skip	. space　size {，value}	用 value 填充 size 个字节，value 默认为 0
. ltorg	. ltorg	声明一个数据缓冲池

【例 3-20】数据定义伪操作使用示例。

```
. byte   20,0xFF,'A'             @分配一段字节内存单元,并用 20，0xFF,'A'初始化
. long   0x12345678,23545        @分配一段字内存单元,并用 0x12345678,23545 初始化
. ascii  "Testing"               @定义一个非 0 结束符字符串"Testing"
. fill   10,2,0x1111             @分配一段内存单元,并用两字节数据 0x1111 填充 10 次
```

【例 3-21】声明一个数据缓冲池来存储 0x12345678。

```
LDR R0,=0x12345678
ADD R1,R1,R0
B       Handler
. ltorg                     @声明文字池,此地址存储 0x12345678
```

注：.ltorg 伪操作在子程序返回或无条件转移指令后使用，这样处理器不会将缓冲池中的内容当做指令来执行。

3. 代码控制伪操作

GNU 编译环境下的代码控制伪操作如表 3-23 所示。

表 3-23 代码控制伪操作

伪操作符	语法格式	说明
. section	. section expr	定义域中包含的段
. text	. text {subsection}	将操作符开始的代码编译到代码段或代码段子段
. data	. data {subsection}	将操作符开始的数据编译到数据段或数据段子段
. bss	. bss {subsection}	将变量存放到. bss 段或. bss 段的子段
. code 16/. thumb	. code 16 / . thumb	表明当前汇编指令的指令集选择 Thumb 指令集
. code 32/. arm	. code 32 / . arm	表明当前汇编指令的指令集选择 ARM 指令集
. align/. balign	.align {alignment} {，fill} {，max}	通过添加填充字节使当前位置满足一定的对齐方式
. end	. end	标记汇编文件的结束行，即标号后的代码不作处理
. org	. org offset {，expr}	指定从当前地址加上 offset 开始存放代码，并且从当前地址到当前地址加上 offset 之间的内存单元，用零或指定的数据进行填充

【例 3-22】代码控制伪操作使用示例。

```
. global     _start               @定义全局标号_start
. include    "2410addr.inc"       @包含外部文件 2410addr.inc
. equ        NUM, 0x1f
. text                            @声明代码段开始
. arm                             @声明 32 位 arm 指令
_start:
        MOV R0, #NUM              @R0=0x1f
        …
. end                             @文件结束
```

4. 预定义控制伪操作

汇编器在对程序代码进行编译时，会根据预定义控制伪操作的定义情况对程序进行编译，常用的由文件包含、条件编译和宏定义，如表 3-24 所示。



表 3-24　预定义控制伪操作

伪操作符	语法格式	说明
. include	. include "filename"	将一个源文件包含到当前源文件中
. macro . exitm . endm	. macro {macroname {parameter {, parameter} …} … . endm	. macro 伪操作标识宏定义的开始，. endm 标识宏定义的结束。用. macro 及. endm 定义一段代码，称为宏定义体 . exitm 伪操作用于提前退出宏
. if . else . endif	. if　condition … {. else …} . endif	当满足某条件时对一组语句进行编译，而当条件不满足时则编译另一组语句。其中. else 可以缺省

【例 3-23】条件编译。

```
    .if val1 < 5
            MOV      R0,#0                @条件成立则 R0=0
    .else
            MOV      R1,#0                @条件不成立则 R1=0
    .endif
```

【例 3-24】带参数的宏定义。

```
    .macro  HANDLER HandleLabel           @宏定义
            sub      sp,sp,#4
            stmfd     sp!,{r0}
            ldr      r0, =\HandleLabel    @宏字符参数可使用"\字符"直接使用
            …
    .endm                                 @宏定义结束
```


3.5　ARM 汇编语言程序设计实例

一般来说，在嵌入式系统编程中，系统启动、初始化代码必须用汇编语言来编写。本节将介绍若干 ARM 汇编语言编程实例，意在帮助读者为嵌入式汇编语言编程打下良好基础。

【例 3-25】实现将寄存器高位和低位对称互换。

将一个寄存器的第 0 位和第 31 位互换，第 1 位和第 30 位互换，第 2 位和第 29 位互换，……，第 15 位和第 16 位互换。

解：可通过移位的方法依次从低位到高位取出源数据的各个位，然后将其放置到目标寄存器的最低位，再通过移位操作，送入相应位，程序代码如下。

1) GNU 编译环境下：

```
. global    _start                      @定义全局标号_start
. text                                  @声明代码段开始
. arm                                   @声明 32 位 arm 指令
_start:
        LDR    R0, = 0x12345678         @输入数据
        MOV    R2, #0                   @目标寄存器清 0
exchange:
        MOV    R1, #32                  @计数器
bit_shift:
        AND    R3, R0, #1           @取出源数据最低位送 R3
        ORR    R2, R3, R2, LSL #1   @目标数据左移 1 位，并将取出数据送至其最低位
        MOV    R0, R0, LSR #1       @源数据右移 1 位
        SUBS   R1, R1, #1           @计数值减 1
        BNE    bit_shift            @若互换未完成，继续
stop:
        B       stop
. end
```

2) ADS 编译环境下：

```
    AREA bit_exc,CODE,READONLY    ;定义了一个只读的叫 bit_exc 的代码段
    ENTRY                         ;程序入口
    CODE32                        ;声明 32 位 arm 指令
START
    LDR      R0,= 0x12345678      ;输入数据
    MOV      R2,#0                ;目标寄存器清 0
Exchange
    MOV      R1,#32               ;计数器
Bit_shift
    AND      R3,R0,#1             ;取出源数据最低位送 R3
    ORR      R2,R3,R2,LSL #1      ;目标数据左移 1 位,并将取出数据送至其最低位
    MOV      R0,R0,LSR #1         ;源数据右移 1 位
    SUBS     R1,R1,#1             ;计数值减 1
    BNE      Bit_shift            ;若互换未完成,继续
Stop
    B        Stop
    END
```


【例 3-26】实现字符串复制。

编写 ARM 汇编程序，实现将字符串“Testing!”复制到目标地址处。

解：在 GNU ARM 开发环境下编程，采用“. string”伪操作定义字符串，每个字符占 1 个字节，而且字符串自动添加 0 结束符，所以在存数/取数时采用 LDRB/STRB 指令，并



且可以根据 0 结束符判断是否到达字符串末尾，程序代码如下。

1) GNU 编译环境下：

```
. global    _start                         @定义全局标号_start
. text                                     @声明代码段开始
. arm                                      @声明 32 位 arm 指令
_ start:
        LDR     R0,= srcstr                @源字符串指针
        LDR     R1,= dststr                @目标字符串指针
strcopy:
        LDRB    R2,[R0],#1                 @逐个复制字符串
        STRB    R2,[R1],#1
        CMP     R2,#0
        BNE     strcopy                    @判断是否已到字符串末尾
stop:
        B       stop
srcstr:
        . string "Testing! "
dststr:
        . string " "
. end
```

2) ADS 编译环境下：

```
        AREA str_copy,CODE,READONLY        ;定义了一个只读的叫 str_copy 的代
                                            码段
        ENTRY                              ;程序入口
        CODE32                             ;声明 32 位 arm 指令
START
        LDR     R0,= Srcstr                ;源字符串指针
        LDR     R1,= Dststr                ;目标字符串指针
Strcopy
        LDRB    R2,[R0],#1                 ;逐个复制字符串
        STRB    R2,[R1],#1
        CMP     R2,#0
        BNE     Strcopy                    ;判断是否已到字符串末尾
Stop
        B       Stop
Srcstr
        DCB     "Testing! ",0
Dststr
        DCB     " "
        END
```

【例 3-27】用多寄存器传送指令 LDM/STM 实现内存数据区块复制操作。

请用 ARM 指令编写程序，实现将数据(20 个字)从源数据区 src 复制到目标数据区 dst，要求以 8 个字为单位进行块复制，如果不足 8 个字时则以字为单位进行复制。

解：首先应计算以 8 个字为单位的块复制的次数和单字复制的次数，在使用寄存器组时应注意保护现场，程序代码如下。

1) GNU 编译环境下：

```
. global     _start
. equ        NUM,      20                        @设置要复制的字数
. text
. arm
_start:
             LDR       R0,= src                  @设置源数据区指针 R0
             LDR       R1,= dst                  @设置目标数据区指针 R1
             MOV       R2,#NUM                   @字单元个数
             MOV       SP,#0x9000
             MOVS      R3,R2,LSR #3              @获得块复制次数
             BEQ       copy_words
             STMFD     SP! ,{R4-R11}             @保存要使用的寄存器 R4-R11
copy_8words:
             LDMIA     R0!,{R4-R11}              @从源数据区复制 8 个字
             STMIA     R1!,{R4-R11}              @保存到目的数据区
             SUBS      R3,R3,#1
             BNE       copy_8words
             LDMFD     SP!,{R4-R11}              @恢复寄存器组
copy_words:
             ANDS      R2,R2,#7                  @获得要复制的剩余字个数
             BEQ       stop                      @判断剩余字数是否为 0
word_copy:
             LDR       R3,[R0],#4                @从源数据区复制一个字
             STR       R3,[R1],#4                @保存到目的数据区
             SUBS      R2,R2,#1
             BNE       word_copy
stop:
             B         stop
. ltorg
src:
             . long        1,2,3,4,5,6,7,8,9,0x0A,0x0B,0x0C,0x0D,0x0E,
                           0x0F,0x10,0x11,0x12,0x13,0x14
dst:
             . long        0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
. end
```



2) ADS 编译环境下：

```
NUM             EQU   20                            ;设置要复制的字数
                AREA  words_copy,CODE,READONLY      ;定义代码段
                ENTRY                               ;程序入口
                CODE32                              : 声明 32 位 arm 指令
START
                LDR     R0,= src                    ;设置源数据区指针
                LDR     R1,= dst                    ;设置目标数据区指针
                MOV     R2,#NUM                     ;字单元个数
                MOV     SP,#0x9000
                MOVS    R3,R2,LSR #3                ;获得块复制次数
                BEQ     Copy_words
                STMFD   SP! ,{R4-R11}               ;保存要使用的寄存器 R4-R11
Copy_8words
                LDMIA   R0!,{R4-R11}                ;从源数据区复制 8 个字
                STMIA   R1!,{R4-R11}                ;保存到目的数据区
                SUBS    R3,R3,#1
                BNE     Copy_8words
                LDM     SP!,{R4-R11}                ;恢复寄存器组
Copy_words
                ANDS    R2,R2,#7                    ;获得要复制的剩余字个数
                BEQ     stop                        ;判断剩余字数是否为 0
Word_copy
                LDR     R3,[R0],#4                  ;从源数据区复制一个字
                STR     R3,[R1],#4                  ;保存到目的数据区
                SUBS     R2,R2,#1
                BNE      Word_copy
Stop
                B        Stop
                LTORG
src
                DCD  1,2,3,4,5,6,7,8,9,0xA,0x0B,0x0C,0x0D,0x0E,0x0F,0x10,
                     0x11,0x12,0x13,0x14
dst
                DCD  0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
                END
```


3.6 案 例 分 析

本章导入案例中介绍了 ARM 启动代码的功能及处理流程。启动代码是系统上电或复位



后首先执行的一段代码，由于启动过程的执行操作都与处理器硬件相关，所以都必须使用汇编语言实现。下面从本章所学的ARM指令集的角度对启动代码的部分内容作简要分析。

1. 中断向量表

因为每个中断只占据向量表中1个字的存储空间，只能放置一条ARM分支指令，使程序跳转到存储器的其他地方，再执行中断处理。

中断向量表的程序实现：

```
AREA    Boot, CODE, READONLY          ;定义了一个只读的名字为Boot的代码段
ENTRY                                 ;进入该代码段
B   ResetHandler                      ;跳转到复位入口
B   UndefHandler                      ;跳转到未定义异常入口
B   SWIHandler                        ;跳转到SWI异常入口
B   PreAbortHandler                   ;跳转到指令中止异常入口
B   DataAbortHandler                  ;跳转到数据中止异常入口
B                                     ;保留
B   IRQHandler                        ;跳转到IRQ中断入口
B   FIQHandler                        ;跳转到快速中断入口
```

其中关键字ENTRY是链接的时候要确保这段代码被链接在0地址处，并且作为整个程序的入口。

2. 堆栈初始化

因为ARM有七种运行模式，每一种模式的堆栈指针寄存器(SP)都是独立的。因此，对程序中需要用到的每一种模式都要给SP定义一个堆栈地址。方法是改变状态寄存器内的状态位，使处理器切换到不同的状态，然后给SP赋值。

这是一段堆栈初始化的代码示例，其中只定义了未定义指令中止模式的SP指针：

```
MRS     R0,CPSR
BIC     R0,R0,#MODEMASK              ;将CPSR模式控制位清0(MODEMASK=0x1f)
ORR     R1,R0,#UNDEFMODE|NOINT       ;UNDEFMODE|NOINT=0x1b|0xc0
MSR     CPSR_cxfs,R1
LDR     SP,=UndefStack               ;初始化未定义堆栈指针
```

其中，使用MRS和MSR指令实现了CPSR寄存器的读—修改—写操作；利用逻辑运算指令BIC和ORR实现将CPSR模式位设置为未定义指令中止模式，并关闭IRQ和FIQ中断，但不影响其他位的值；利用LDR伪指令给堆栈指针SP赋值。

3. 应用程序执行环境初始化

所谓应用程序执行环境的初始化，就是完成必要的从ROM数据区到RAM的RW区数据传输以及整个ZI区清零。

下面是在ADS下，一种常用存储器模型的直接实现：

```
    LDR    r0,=|Image$$RO$$Limit|      ;r0为ROM数据区首地址
```



```
        LDR    r1,=|Image$$RW$$Base|        ;r1 指向 RAM 的 RW 区首地址
        LDR    r3,=|Image$$ZI$$Base|        ;ZI 区在 RAM 里面的起始地址
        CMP    r0,r1                        ;比较它们是否相等
        BEQ    %F1                          ;地址重合表示没有 ROM 数据复制到 RW 区
0       CMP    r1,r3                        ;完成 ROM 数据区向 RW 区数据区的复制
        LDRCC r2,[r0],#4
        STRCC r2,[r1],#4
        BCC    %B0                          ;r1 小于 r3 则继续循环复制
1       LDR    r1,=|Image$$ZI$$Limit|       ;r1 指向 ZI 区的结束地址
        MOV    r2,#0
2       CMP    r3,r1                        ;循环给 ZI 数据区写 0
        STRCC r2,[r3],#4
        BCC    %B2                          ;若 r3 小于 r1 则继续循环写 0
```

其中，使用 CMP 比较指令和带 EQ(相等)条件后缀的 B 指令使程序产生分支；使用 CMP 指令和 CC(无符号数小于)后缀构成循环结构，这也是 ARM 汇编语言循环结构的一般编写方法。可以看出，ARM 指令条件码的使用可以使程序结构紧凑清晰，几乎所有的 ARM 指令都可使用条件码后缀。

4. 呼叫主应用程序

当所有的系统初始化工作完成之后，就需要把程序流程转入主应用程序。最简单的一种情况如下：

```
IMPORT   _main       ;IMPORT 表示_main 在外部定义
B        _main       ;转向 C 语言的主函数
```

直接从启动代码跳转到应用程序的主函数入口，主函数名字可以由用户定义。

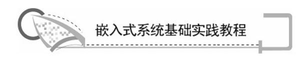
本 章 小 结

指令是汇编语言程序设计的基础，在基于 ARM 的嵌入式软件开发中，即使大部分程序使用高级语言完成，但系统启动、引导代码仍必须用汇编语言来编写。本章介绍了 ARM 指令集的语法格式、ARM 处理器的寻址方式，分类介绍了 ARM 指令、伪指令及伪操作的详细功能以及在使用中的注意事项，并给出汇编语言程序设计的实例分析。

(1) ARM 指令集概述：简要介绍了 ARM 指令系统的特点、ARM 指令集的基本格式和分类情况，最后介绍了 ARM 指令条件执行的含义。

(2) ARM 处理器的寻址方式：寻址方式是根据指令中给出的地址码字段来寻找真实操作数地址的方式，介绍了目前 ARM 指令系统支持的八种基本的寻址方式。

(3) ARM 指令集：ARM 微处理器指令集可以分为数据处理指令、Load/Store 指令、分支指令、程序状态寄存器访问指令、协处理器操作指令和异常产生指令六大类；分别介绍了各类指令的详细功能及应用。



(4) ARM伪指令及伪操作：在进行汇编语言程序设计时，除了指令之外，还需要掌握ARM汇编伪指令和伪操作。简要介绍了ARM伪指令、ADS编译环境下的伪操作及GNU编译环境下的伪操作的功能及使用方法。

(5) ARM汇编语言程序设计实例：给出若干ARM汇编语言应用实例，为后续嵌入式软件开发打下基础。



ARM 协处理器

ARM微处理器可支持多达16个协处理器，用于各种协处理操作。在程序执行的过程中，每个协处理器只执行针对自身的协处理指令，忽略ARM处理器和其他协处理器的指令。ARM的协处理器指令主要用于ARM处理器初始化ARM协处理器的数据处理操作，在ARM处理器的寄存器和协处理器的寄存器之间传送数据，以及在ARM协处理器的寄存器和存储器之间传送数据。当一个协处理器硬件不能执行属于它的协处理器指令时，将产生一个未定义指令异常中断，在该异常中断处理程序中，可以通过软件模拟该硬件操作。协处理器也能通过提供一组专门的新指令来扩展指令集。例如，有一组专门的指令可以添加到标准ARM指令集中，以处理向量浮点(VFP)运算。

Samsung公司的S3C2410处理器芯片的内核为ARM920T。ARM920T核含有内部协处理器CP15，即通常所说的系统控制协处理器(System Control Coprocessor)，它负责完成大部分的存储系统管理，提供附加的寄存器用来配置和控制Cache、MMU、保护系统、时钟模式及其他的系统选项，如大小端操作。

CP15包含16个32位寄存器，其编号为0～15。实际上对于某些编号的寄存器可能对应多个物理寄存器，在指令中指定特定的标志位来区分这些物理寄存器。这种机制有些类似于ARM中的寄存器，当处于不同的处理器模式时，某些相同编号的寄存器对应于不同的物理寄存器。CP15中的寄存器可能是只读的，也可能是只写的，还有一些是可读可写的。CP15中的寄存器功能如表3-25所示。

表3-25　ARM处理器中CP15协处理器的寄存器

Register(寄存器)	Read	Write
C0	ID Code (1)	Unpredictable
C0	Cache Type (1)	Unpredictable
C1	Control	Control
C2	Translation table base	Translation table base
C3	Domain access control	Domain access control
C4	Unpredictable	Unpredictable
C5	Fault status(2)	Fault status (2)
C6	Fault address	Fault address
C7	Unpredictable	Cache operations
C8	Unpredictable	TLB operations
C9	Cache lockdown(2)	Cache lockdown (2)
C10	TLB lock down(2)	TLB lock down(2)



续表

Register(寄存器)	Read	Write
C11	Unpredictable	Unpredictable
C12	Unpredictable	Unpredictable
C13	Process ID	Process ID
C14	Unpredictable	Unpredictable
C15	Test configuration	Test configuration

说明：

上表中，(1) 表示在寄存器位置 0 可以访问两个寄存器，具体访问哪一个寄存器取决于协处理器指令中第二个操作码的值。

(2) 表示分指令寄存器和数据寄存器。

Unpredictable：从这个写方式读出的值可以是任意的。

只能在特权模式下使用 MRC 和 MCR 指令访问 CP15 的寄存器。

习　　题

一、选择题

1．指令 LDR R0，[R1，#4]! 实现的功能是(　　)。

A．R0←[R1+4]　　B．R0←[R1+4]，R1←R1+4

C．R0←[R1]，R1←R1+4　　D．R0←[R1]，R1←R1−4

2．AND　R6，R7，#0xFF 指令中第 2 操作数#0xFF 的寻址方式是(　　)。

A．寄存器寻址　　B．寄存器间接寻址

C．立即寻址　　D．直接寻址

二、判断题

1．在用户模式下，执行“MOVS　PC，LR”指令可实现异常返回。(　　)

2．CMN 指令与 ADDS 指令的区别在于 CMN 指令不保存结果。(　　)

3．LDM 指令中寄存器和连续内存单元的对应关系：编号高的寄存器对应内存低地址单元。(　　)

三、问答题

1．简述 ARM 指令集的分类情况。

2．BIC 指令有什么作用？

3．ARM 指令中的第二操作数有几种形式？列出三个合法的立即数。

4．请指出 MOV 指令和 LDR 加载指令的区别及用途。

5．ARM 汇编伪指令和伪操作的区别是什么？

6．写出一段汇编语言程序，实现将处理器模式切换到未定义指令中止模式，并关闭中断。



7．如何实现两个 64 位数的加法和减法操作？

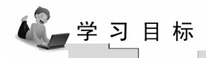
8．在 GNU ARN 环境下，请用 ARM 指令编写程序，实现将数据(18 个字)从源数据区 src 复制到目标数据区 dst，要求以四个字为单位进行块复制，如果不足四个字时则以字为单位进行复制。

内存数据区定义如下：

```
src:
    DCD     1,2,3,4,5,6,7,8,9,0xA,0xB,0xC,0xD,0xE,0xF,0x10,0x11,0x12
dst:
    DCD     0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
```

第 4 章

嵌入式系统硬件平台



学 习 目 标

了解嵌入式系统硬件平台的基本组成;
熟悉 S3C2410X 处理器的架构和特点;
理解以嵌入式微处理器为核心的接口扩展方法。

知 识 结 构



图 4.1　嵌入式系统硬件平台知识结构图



导入案例

构建一个以 ARM 为核心的应用系统。

前面章节介绍了 ARM 体系结构和指令系统，为构建嵌入式系统应用奠定了基础。但是，ARM 只是 IP 核，要投入实际应用，尚需借助于芯片乃至板级系统。在嵌入式系统学习的初期阶段，通常需要依赖于实验系统进行微小应用开发，以熟悉嵌入式系统开发流程。

图 4.2 是武汉创维特信息技术有限公司的一款 JXARM9-2410-1 教学实验箱，该实验箱为 ARM 核工作提供了工作平台，具有嵌入到应用环境的多数接口，如 RS232/RS485 串行接口、以太网接口、USB 接口、CF 卡接口、IDE 接口、MMC/SD 卡接口、IIS 接口、I^2C 接口、CAN 总线、A/D 及 D/A 转换接口、标准计算机打印口、彩色 LCD 显示器加触摸屏、4×4 键盘、PS/2 键盘和鼠标接口等。

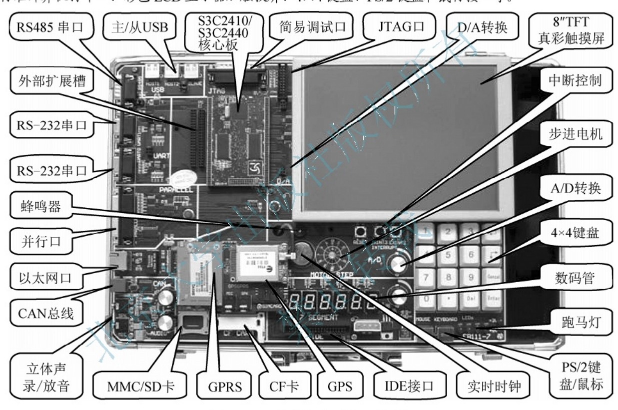


图 4.2　一个嵌入式系统硬件板卡

JXARM9-2410 教学实验系统的硬件部分包括核心模块、调试模块、通信模块、人机交互模块、IDE/CF/SD/MMC 接口模块、A/D 及 D/A 模块、工业控制模块、GPRS 模块、GPS 模块和扩展模块。

1. 核心模块

1) SDRAM 存储器；

2) Flash 存储器；

3) 串行通信口；

4) I^2S、I^2C 总线接口。

2. 调试模块

1) 标准 JTAG 接口；

2) 简易 JTAG 调试接口。

3. 通信模块

1) 以太网接口；

2) USB 接口；
3) 标准计算机打印口。
4. 人机交互模块
1) 显示器/触摸屏；
2) 按键；
3) PS/2 键盘和鼠标接口；
4) USB 鼠标和键盘接口。
5. IDE/CF/SD/MMC 接口模块
1) 标准 IDE 硬盘接口；
2) 标准 CF 卡接口；
3) SD/MMC 卡接口。

前面以 JXARM9-2410-1 实验箱为例，介绍了实际应用系统中的硬件平台应具备的基本要素。本章着眼于嵌入式系统的硬件平台，以最小系统为突破口，讲解以 ARM 为核心的微处理器芯片与外围设备的接口连接，逐步形成能够实际应用的硬件系统，为后续的软件编程奠定良好基础。

4.1　嵌入式最小系统

4.1.1　最小系统架构

嵌入式最小系统即是在尽可能减少上层应用的情况下，能够使系统运行的最小化模块配置。对于一个典型的嵌入式最小系统，以 ARM 处理器为例，其构成模块及其各部分功能如图 4.3 所示，其中 ARM 微处理器、Flash 和 SDRAM 模块是嵌入式最小系统的核心部分。各模块有以下基本功能。

- 时钟模块：通常经 ARM 内部锁相环进行相应的倍频，以提供系统各模块运行所需的时钟频率输入；
- Flash 存储模块：存放启动代码、操作系统和用户应用程序代码；
- SDRAM 模块：为系统运行提供动态存储空间，是系统代码运行的主要区域；
- JTAG 模块：实现对程序代码的下载和调试；
- UART 模块：实现对调试信息的终端显示；
- 复位模块：实现对系统的复位。

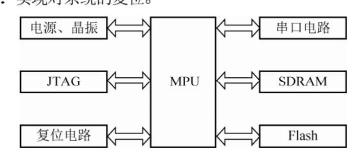


图 4.3　嵌入式最小系统结构框图

4.1.2 模块典型电路

以 S3C2410X 微处理器为例，构建嵌入式最小系统。下面是部分模块的常见参考电路。

1. 电源电路

S3C2410X 工作时内核需要 1.8V 电压，I/O 端口和外设需要 3.3V 电压。VDDi/ VDDiarm 引脚是供 S3C2410X 内核的 1.8V 电压；VDDalive 引脚是功能复位和端口状态寄存器电压。M12 引脚 RTCVDD 是 RTC 模块的 1.8V 电压，用电池供电保证系统的掉电后保持实时时钟。VDDOP 引脚是 I/O 端口 3.3V 电压；VDDMOP 引脚是存储器 I/O 端口电压；还有一系列 VSS 引脚需要接到电源地上。3.3V 电压从 5V 用 AMS1117-3.3 转换得到如图 4.4(a)所示；1.8V 从 3.3V 通过 MIC5207-1.8 转换得到，如图 4.4(b)所示。

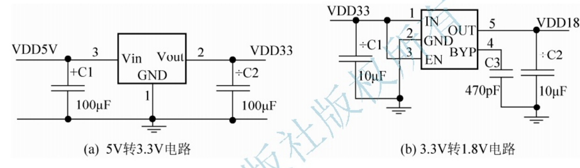


图 4.4 电源电路

2. 晶振电路

S3C2410X 内部有时钟管理模块，有两个锁相环，其中 MPLL 能够产生 CPU 主频 FCLK、AHB 总线外设时钟 HCLK 和 APB 总线外设时钟 PCLK；UPLL 产生 USB 模块的时钟。OM3、OM2 都接地时，主时钟源和 USB 模块时钟源都由外接晶振产生。在 XTIp11 和 XTOp11 之间连接主晶振，可以选择 12MHz 晶振，通过内部寄存器的设置产生不同频率的 FCLK、HCLK 和 PCLK；在 XTIrtc 和 XTOrtc 上需要接 32.768 kHz 的晶振供 RTC 模块使用。同时在 MPLLCAP 和 UPLLCAP 上也要外接 5pF 的环路滤波电容。晶振电路如图 4.5 所示。

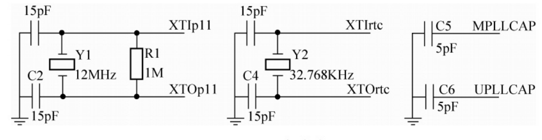


图 4.5 晶振电路

3. 复位电路

S3C2410X 的 J12 引脚为 nRESET 复位引脚，nRESET 上给四个 FCLK 时间的低电平后就可以复位。可以设计如图 4.6 所示的复位电路，其中上电复位是靠 RC 电路特性完成，开关二极管 1N4148 在手动复位时对电容起快速放电的作用，因此可以把复位电平快速拉到 0V。反响门 74HC14 可以起到延时作用，保证有足够的复位时间。

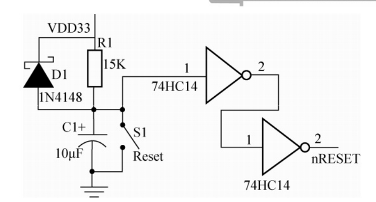


图 4.6　复位电路

4.2　S3C2410X 微处理器

4.2.1　S3C2410X 微处理器简介

三星公司推出的 16/32 位 RISC 处理器 S3C2410X 为手持设备和一般类型应用提供了低价格、低功耗、高性能小型微控制器的解决方案。S3C2410X 基于 ARM920T 内核，0.18μm 工艺的 CMOS 标准宏单元和存储器单元，采用了高级微控制器总线(Advanced Microcontroller Bus Architecture，AMBA)的新型总线结构，提供了丰富的片上资源，特别适用于对成本和功耗敏感的应用。

S3C2410X 微处理器提供了丰富的内部设备：双重分离的 16KB 的指令缓存和 16KB 数据缓存、MMU 虚拟存储器管理部件、LCD 控制器、支持 NAND Flash 系统引导、外部存储控制器、3 通道 UART、4 通道 DMA、4 通道 PWM 定时器、I/O 端口、定时器、8 通道 10 位 A/D 转换器、触摸屏接口、I^2C 总路线接口、USB 主机端口、USB 设备端口、SD 主卡及 MMC 卡接口、2 通道 SPI 以及内部 PLL 时钟倍频器。

S3C2410X 微处理器的特征如表 4-1 所示。

表 4-1　S3C2410X 微处理器特征

特点	描述
ARM920T 架构	采用 ARM920T 作为处理器内核，因此具有 ARM920T 的所有特征
双重分离缓存	具有 64 项全相连模式，采用指令和数据双重分离的缓存技术，具有 16KB 指令缓存及 16KB 数据缓存
存储管理部件	内部集成了存储管理部件(MMU)，配合需要 MMU 支持的嵌入式操作系统
外部存储控制器	支持大/小端格式，寻址空间每 Bank128MB(共 1GB)，支持可编程的每 Bank 8/16/32 位数据总线宽度
LCD 控制器	最大支持 4 位双扫描、4 位单扫描及 8 位单扫描三种类型的 STN LCD 显示屏。支持单色模式 4 级、16 级灰度的 STN LCD，256 色和 4096 色 STN LCD；支持 640×480、320×240、160×160 等不同尺寸的 LCD。256 色模式下支持的最大虚拟屏是 4096×1024、2048×2048 及 1024×4096
TFT 彩色显示屏	支持彩色 TFT 的 1、2、4 或 8bbp 调色显示，支持 16bbp 无调色显示，在 24bbp 模式下支持最大 16M 色的 TFT，支持不同尺寸的液晶屏


续表

特点	描述
DMA 控制器	具有 4 通道的 DMA 控制器，支持存储器到存储器，I/O 到存储器，存储器到 I/O 以及 I/O 到 I/O 的数据传输，采用猝发传输模式加快传输速率
UART	具有 3 通道 UART，可支持 DMA 模式和中断模式工作，支持 5～8 位的串行数据格式，支持外部时钟作为 UART 的时钟，可编程，支持红外 IrDA1.0，具有测试用的自发自收模式
I^2C 总线及 I^2S 总线接口	具有 1 通道多主 I^2C 总线，支持 8 位串行双向数据传输，标准模式下传输速率可达 100 kbit/s，快速模式下传输速率可达 400 kbit/s。S3C2410X 具有 1 通道音频 I^2S 总线接口，可基于 DMA 方式工作，可采用 I^2S 格式和 MSB-justified 数据格式，支持 8 位/16 位串行数据传输
定时器	有 4 通道 16 位 PWM 定时器、1 通道 16 位通用定时器及 16 位看门狗定时器各 1 个
SPI 接口	兼容并包括通道(SPI)协议 2.11 版本，发送与接收具有 2×8 位的移位寄存器，可基于 DMA 或中断模式工作
通用 I/O 端口	具有 117 个通用多功能 I/O 端口，其中有 24 个具有外部中断功能
RTC	内部集成了可以进行锁相环控制的时钟发生器，使系统可以灵活地控制适中信号的发生；内置的 RTC 模块自带日历功能，使系统在使用日历功能时可直接读取相应寄存器的值
触摸屏接口	具有 8 通道多路复用的 10 位 ADC，最大采样率为 500kSPS
SD 主机接口	兼容 SD 存储卡协议 1.0 版，兼容 SDIO 卡协议 1.0 版，接收和发送均具有 FIFO，基于 DMA 或中断模式工作，兼容 MMC 卡协议 2.11 版
USB 主机及 USB 设备	支持 2 个端口的 USB 主机和 1 个端口的 USB 设备，使系统与 USB 设备的信息交换更加方便快捷
中断控制器	有 55 个中断源，包括 1 个看门狗定时器、5 个定时器、9 个 UART、24 个外部中断、4 个 DMA、2 个 RTC、2 个 ADC、1 个 I^2C、2 个 SPI、1 个 SDI、2 个 USB、1 个 LCD 及 1 个电池故障。外部中断源可编程为电平和边缘触发，触发电平可编程，支持快速中断服务
额定工作参数	内核电压分别为 1.8V，I/O，存储器电压均为 3.3V，其中 S3C2410X 的工作频为 203MHz 或 266MHz

4.2.2 S3C2410X 微处理器体系结构

S3C2410X 微处理器内部体系结构如图 4.7 所示。S3C2410X 采用 ARM920T 核，而 ARM920T 又集成了 ARM9TDMI，是中高档 32 位嵌入式微处理器，由于采用 ARM920T 体系结构，因此内部具有分离的、16KB 大小的指令缓存和 16KB 大小的数据缓存，同时，S3C2410X 采用哈佛体系结构，将程序存储器与数据存储器分开，加入了 MMU，采用五级指令流水线。ARM920T 核包含系统控制协处理器 CP15，CP15 协处理器提供附加的寄存器用来配置和控制 Cache、MMU、保护系统、时钟模式以及其他系统选项(如大小端操作)。程序员可以使用协处理器操作指令访问定义在 CP15 中的寄存器。使用 ARM 公司特

有的 AMHA 总线，如图 4.8 所示。对于高速设备采用 AHB 总线，而对于低速内部外设则采用 APB 总线。AHB 通过桥接器转换成 APB。

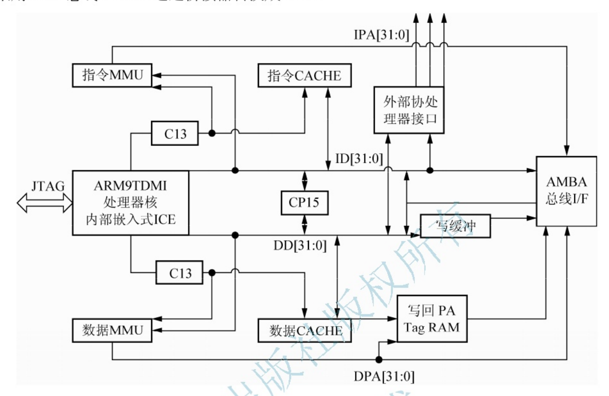


图 4.7　ARM920T 核结构示意图

地址桥为 ARM920T 核提供了透明访问每一个连接在 APB 上外设的路径。桥是一个 AHB 的从设备，为高速的 AHB 和低功耗的 APB 提供了接口。AHB 上的读写传输会转换为 APB 上等量传输。由于 APB 不是流水线方式，因此当 AHB 必须等待 APB 时，从 APB 输出与输入传输必然会增加等待状态。

桥包括以下几个模块：

1) AHB 从总线接口；

2) 独立于器件存储器映射的 APB 转移状态机；

3) APB 输出信号发生器。

APB 桥对当前 AHB 管理器的请求做出响应，将 AHB 事务转换为 APB 事务。当访问一个不能识别的地址时，系统仍然正常运行，但是不会去选择外围设备。

通过 AMBA 总线，S3C2410X 内部集成了许多外设接口，包括了 S3C44B0X 所有内部外设，还增加了许多新外设接口，主要的内部外设包括与 AHB 总线相连的高速接口，如 LCD 接口、USB 接口、中断控制接口、电源管理接口、存储器接口、Boot Loader 接口，与 APB 总线相连的低速接口，如三个通用异步通信接口(UART0、UART1、UART2)、SDI/MMC 接口、看门狗定时器、总线控制器、两个 SPI 接口、四个 PWM 定时器、实时钟、通用并行端口、I^2C 总线接口以及 I^2S 总线接口等。

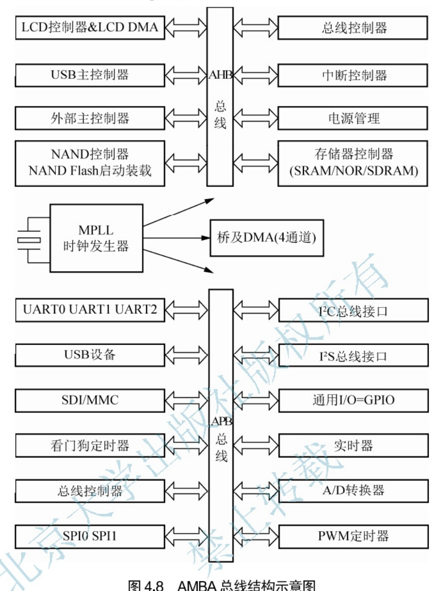


图 4.8　AMBA 总线结构示意图

4.3　存 储 系 统

4.3.1　存储系统基础知识

1. 存储器的存储容量与速度比较

计算机系统期望存储器容量大、存取速度快、成本价格低。这些要求本身是相互矛盾的，也是相互制约的，在同一存储器中难以同时满足。图 4.9 是几种不同存储器的容量、速度的层次关系。

在图 4.9 中，位于上层的存储体比位于下层的存储体访问速度快，但通常存储容量没有下层存储体大。通常，半导体存储器具有较快的存取速度，但存储容量有限；磁盘存储容量大，但存取速度慢。为了发挥它们各自的优势，按照一定的体系结构有机地组合起来，即可得到一个分级存储结构的存储系统。



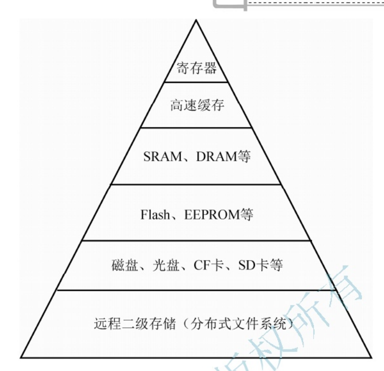


图 4.9　存储体访问速度层次图

2. *存储系统的层次结构*

合理地分配存储容量、速度和成本的有效措施是实现分级存储。这是一种把几种存储技术结合起来、互相补充的折中方案。图 4.10 是一个三级存储体系的结构图。从中可看出这个层次结构的规律(从左到右)：

- 价格逐次降低；
- 容量依次增加；
- 访问时间逐渐增大。

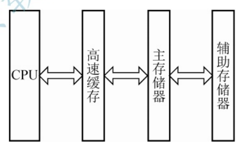


图 4.10　三级存储体系结构示意图

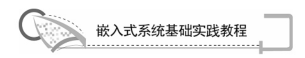
使用三层存储体系后，从 CPU 的角度看，存储速度接近于高速缓冲存储器，容量及成本却又接近辅助存储器，这样使系统获得较高的性能价格比。在三级存储体系中，各级存储器中存放的信息必须满足以下两个原则。

1) 一致性原则：同一个信息会同时存放在多个级别的存储器中，而这一信息在多个级别的存储器中必须保持相同的值。

2) 包含性原则：处在内层(靠近 CPU)的存储器中的信息一定包含在各外层的存储器中，即内层存储器中的信息一定是各外层存储器中所存信息的子集，这是保证程序正常运行、实现信息共享、提高系统资源利用率的必要条件，反之则不成立。

分层结构的存储体系的理论基础是程序局部性原理，该原理主要体现在以下两个方面。

1) 时间方面：在一小段时间内，最近被访问过的程序和数据很可能再次被访问。即当前正在使用的信息很可能是后面立刻就要使用的信息，程序循环和堆栈等操作中的信息便是如此。



2) 空间方面：这些最近被访问过的程序和数据，往往集中在一小片存储区域中。例如，以顺序执行为主流的程序和数据。指令顺序执行比转移执行的可能性要大(大约 5∶1)。

3. MMU

虚拟存储管理(Virtual Memory Management)机制在现代操作系统中广泛使用，其实现需要处理器中的存储管理部件(Memory Management Unit，MMU)提供支持，这里简要介绍 MMU 的作用。

首先引入两个概念：虚拟地址和物理地址。如果处理器没有 MMU，或者有 MMU 但没有启用，如图 4.11(a)所示，CPU 执行单元发出的内存地址将直接传送到芯片引脚，驱动物理内存或设备，这称为物理地址(Physical Address，PA)。如果处理器启用了 MMU，如图 4.11(b)所示，CPU 执行单元发出的内存地址将被 MMU“截获”，从 CPU 到 MMU 的地址称为虚拟地址(Virtual Address，VA)，而 MMU 将这个地址翻译成物理地址发到 CPU 芯片的外部地址引脚上。

读者可以通过下面的例子，加深对虚拟地址、物理地址的理解。图 4.11(b)中，如果是 32 位处理器，则内地址总线是 32 位的，与 CPU 执行单元相连，而经过 MMU 转换之后的外地址总线则不一定是 32 位的。也就是说，虚拟地址空间和物理地址空间是独立的，32 位处理器的虚拟地址空间是 4GB，而物理地址空间既可以大于也可以小于 4GB。

MMU 将 VA 映射到 PA 是以页(Page)为单位的，32 位处理器的页尺寸通常是 4KB。例如，MMU 可以通过一个映射项将 VA 的一页 0xB7001000～0xB7001FFF 映射到 PA 的一页 0x4000～0x4FFF，如果 CPU 执行单元要访问虚拟地址 0xB7001008，那么实际访问到的物理地址是 0x2008。物理内存中的页称为物理页面或者页帧(Page Frame)。虚拟内存的页面与物理内存的页帧之间的映射关系是通过页表(Page Table)来描述的，页表保存在物理内存中，MMU 会查找页表来确定一个 VA 应该映射到什么 PA。

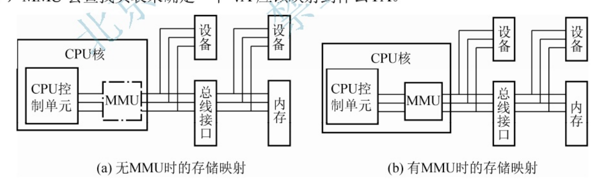


图 4.11　MMU 存储映射示意图

MMU 除了做地址转换之外，还提供内存保护机制。很多体系结构都有用户模式(User Mode)和特权模式(Privileged Mode)之分，操作系统可以在页表中设置每个内存页面的访问权限，访问权限又分为可读、可写和可执行三种。当 CPU 要访问一个 VA 时，MMU 即检查 CPU 当前处于用户模式还是特权模式，访问内存的目的是读数据、写数据还是取指令。如果和操作系统设定的页面权限相符，就允许访问，把它转换成 PA，否则不允许访问，产生异常(Exception)。



通常操作系统把虚拟地址空间划分为用户空间和内核空间，例如，x86 平台的 Linux 系统虚拟地址空间是 0x00000000～0xFFFFFFFF，前 3GB(0x00000000～0xbFFFFFFF)是用户空间，后 1GB(0xc0000000～0xFFFFFFFF)是内核空间。用户程序加载到用户空间，在用户模式下执行，不能访问内核中的数据，也不能跳转到内核代码中执行。这样可以保护内核，如果一个进程访问了非法地址，最坏情况是该进程崩溃，而不会影响到内核和整个系统的稳定性。

4.3.2　S3C2410X 微处理器的存储器接口

1. S3C2410X 的存储器接口

目前的嵌入式应用系统中，通常使用三种存储器接口电路：NOR Flash 接口、NAND Flash 接口和 SDRAM 接口。引导程序既可存储在 NOR Flash 中，也可存储在 NAND Flash 中，而 SDRAM 中存储的是执行中的程序和产生的数据。存储在 NOR Flash 中的程序可直接执行，与在 SDRAM 执行相比速度较慢。存储在 NAND Flash 中的程序，需要复制到 RAM 中去执行。

S3C2410X 的存储系统具有以下主要特征。

1) 支持数据存储的大/小端选择(通过外部引脚进行选择)。

2) 地址空间：具有 8 个存储体，每个存储体可达 128MB，共计可达 1GB。

3) 对存储体的访问位数可变(8 位/16 位/32 位)。

4) 8 个存储体中，Bank0～Bank5 支持 ROM、SRAM；Bank6、Bank7 可支持 ROM、SRAM、SDRAM 等。

5) 7 个存储体的起始地址固定，一个存储体的起始地址可变。

2. 存储映射

S3C2410X 的存储映射如图 4.12 所示，映射关系分两种情况：NOR Flash 启动和 NAND Flash 启动。从哪种存储器启动取决于系统设计，OM[1:0]引脚用于配置启动方式和数据宽度。当 OM[1:0]等于 01b 或 10b 时，系统从 NOR Flash 启动；OM[1:0]等于 00b 时，系统从 NAND Flash 启动。

图 4.12 中，可以看出 S3C2410X 的存储空间被分成 8 个 Bank，每个 Bank 可达 128MB，由片选信号 nGCSx 选择，存储体支持的存储器类型不尽相同，其中 SROM 表示 SRAM 和 ROM。硬件设计员在扩展存储器时，应特别注意每个 Bank 支持的存储器类型、地址范围等。软件程序员在移植 U-boot 时，需要配置 SDRAM 和 Flash 的 Bank 数，这时可查看电路图，依实际情况配置。例如，SDRAM 或者 Flash 只接了一个片选，那么 Bank 数配置为 1 即可。

当系统从 NOR Flash 启动时，NOR Flash 被映射到 0x0000_0000 地址上，因其支持在片执行，系统可从 NOR Flash 上获得启动代码并且直接执行，而内部的 SRAM 被映射到 0x4000_0000 地址上。当系统从 NAND Flash 启动时，内部 SRAM 被映射到 0x0000_0000 上，系统上电后，芯片固化代码会将 NAND Flash 上的前 4KB 启动代码搬运到内部 SRAM 中，然后从内部 SRAM 启动系统。

地址	OM[0:1]=01,10
0xFFFF_FFFF	
	未使用
0x6000_0000	
	特殊功能寄存器区
0x4800_0000	
0x4000_0FFF	
	BootSRAM 4KB
0x4000_0000	
	SROM/SDRAM nGCS7
0x3800_0000	
	SROM/SDRAM nGCS6
0x3000_0000	
	SROM nGCS5
0x2800_0000	
	SROM nGCS4
0x2000_0000	
	SROM nGCS3
0x1800_0000	
	SROM nGCS2
0x1000_0000	
	SROM nGCS1
0x0800_0000	
	SROM nGCS0
0x0000_0000	

未使用NAND Flash作为启动ROM

地址	OM[0:1]=00
0xFFFF_FFFF	
	未使用
0x6000_0000	
	特殊功能寄存器区
0x4000_0FFF	
	未使用
0x4000_0000	
	SROM/SDRAM nGCS7
0x3800_0000	
	SROM/SDRAM nGCS6
0x3000_0000	
	SROM nGCS5
0x2800_0000	
	SROM nGCS4
0x2000_0000	
	SROM nGCS3
0x1800_0000	
	SROM nGCS2
0x1000_0000	
	SROM nGCS1
0x0800_0000	
	引导内部SRAM 4KB
0x0000_0000	

使用NAND Flash作为启动ROM

图 4.12　S3C2410X 存储映射

图 4.12 中，特殊功能寄存器区定义了配置存储器、内部外设等的寄存器。S3C2410X 的数据手册对这些寄存器进行了详细描述，程序员在配置系统时，可对照数据手册查阅头文件 s3c2410.h 中的定义。

4.3.3　S3C2410X 微处理器的存储器配置实例

以 SDRAM 配置为例，介绍 S3C2410X 微处理器存储器的配置方法。系统使用 SDRAM 之前，需要对 S3C2410X 的存储器控制器进行初始化，核心工作是对 SDRAM(这里使用 Bank6)相关的寄存器进行特殊配置，以使 SDRAM 能够正常工作。由于 C 语言程序使用的数据空间和堆栈空间都定位在 SDRAM 上，因此，如果没有对 SDRAM 的正确初始化，系统就无法正确启动。下面介绍与 SDRAM 相关的寄存器配置。

1. BWSCON 寄存器

BWSCON 寄存器主要用来配置外接存储器的总线宽度和等待状态。在 BWSCON 中，除了 Bank0，其他 7 个 bank 都各对应 4 个相关位的配置，分别为 STn，WSn 和 DWn。这里只需要对 DWn 进行设置，例如，SDRAM(Bank6)采用 32 位总线宽度，因此，DW6=10，其他 2 位采用默认值。BWSCON 寄存器在 Bank6 上的位定义如表 4-2 所示。



表 4-2　BWSCON 寄存器在 Bank6 上的位定义

BWSCON	位	描述	初始状态
ST6	[27]	SRAM 在 Bank6 上是否采用 UB/LB。0：不采用 UB/LB(引脚对应 nWBE[3:0]); 1：采用 UB/LB(引脚对应 nBE[3:0])	0
WS6	[26]	Bank6 的 WAIT 状态。0：WAIT 禁止 ； 1：WAIT 使能	0
DW6	[25:24]	Bank6 的数据总线宽度。00：8 位；01：16 位；10：32 位	0

2．BANKCONn 寄存器的设置

S3C2410X 有 8 个 BANKCONn 寄存器，分别对应着 Bank0～Bank7。由于 Bank6～Bank7 可以作为 FP/EDO/SDRAM 等类型存储器的映射空间，因此与其他 Bank 的相应寄存器有所不同，其中 MT 位定义了存储器的类型。BANKCONn 寄存器在 Bank6 和 Bank7 上的位定义如表 4-3 所示。

表 4-3　BANKCONn 寄存器在 Bank6 和 Bank7 上的位定义

BANKCONn	位	描述	初始状态
MT	[16:15]	Bank6 和 Bank7 的存储器类型。00：ROM 或 SRAM；01：FP DRAM；10：EDO DRAM；11：SDRAM	11

MT 的取值又定义该寄存器余下几位的作用。当 MT=11(即 SDRAM 型存储器)时，BANKCONn 寄存器余下的几位定义如表 4-4 所示。Trcd 是从行使能到列使能的延迟，根据 S3C2410X 的 HCLK 频率(100M)及 HY57V561620T-H 的特性，此项取 01，即 3CLKs。SCAN 为列地址线数量，根据 HY57V561620 特性，此项取 01，即 9 位(A0～A8)。

表 4-4　BANKCONn 寄存器的相关位定义

BANKCONn	位	描述	初始状态
Trcd	[3:2]	$\overline{RAS}$ 到 $\overline{CAS}$ 的延时。00：2 时钟；01：3 时钟；10：4 时钟	10
SCAN	[1:0]	列地址位数。00：8 位；01：9 位；10：10 位	00

3．REFRESH 寄存器

SDRAM 是动态，RAM 需要及时刷新。REFRESH 寄存器是 DRAM/SDRAM 的刷新控制器。位定义如表 4-5 所示。

表 4-5　REFRESH 寄存器位定义

REFRESH	位	描述	初始状态
REFEN	[23]	DRAM/SDRAM 刷新使能。0：禁止；1：使能	1
TREFMD	[22]	DRAM/SDRAM 刷新模式。0：CBR/自动刷新；1：自刷新。自刷新时，DRAM/SDRAM 控制线需要适当的电平驱动	0
Trp	[21:20]	DRAM/SDRAM $\overline{RAS}$ 预充电时间	10
Tsrc	[19:18]	SDRAM 半行周期时间。SDRAM 的行周期时间 Trc=Trp+Tsrc	11
Refresh Counter	[10:0]	SDRAM 刷新计数器值。刷新时间=(2^{11}−刷新计数器值+1)/HCLK 如果刷新时间是 15.6μs，HCLK 是 60MHz，刷新时间计算如下： 刷新时间=2048+1−60×15.6=1113	0

4．BANKSIZE 寄存器

BANKSIZE 寄存器包含存储器猝发使能、时钟使能、存储空间映射的配置，详细信息请查看表 4-6。初始化时，BURST_EN 可以取 0 或 1，为了提高效率，最好设置为 1，SCKE_EN 设置为 1，SCLK_EN 设置为 1，BK76MAP 设置为 2。

表 4-6　BANKSIZE 寄存器定义

BANKSIZE	位	描述	初始状态
BURST_EN	[7]	ARM 内核猝发操作使能。0：禁止猝发操作；1：使能猝发操作	0
保留	[6]	未用	0
SCKE_EN	[5]	SCKE 使能控制。0：SDRAM SCKE 禁止；1：SDRAM SCKE 使能	0
SCLK_EN	[4]	SCLK 只有在 SDRAM 访问周期内才有效，这样做是可以减少功耗。当 SDRAM 不被访问时，SCLK 变成低电平。0：SCLK 总是激活；1：SCLK 只有在访问期间激活	0
保留	[3]	未用	0
BK76MAP	[2:0]	BANK6/7 的存储空间分布。010：128MB/128MB；001：64MB/64MB；000：32M/32M；111：16M/16M；110：8M/8M；101：4M/4M；100 = 2M/2M	010

5．MRSR 寄存器

MRSR 寄存器有两个，分别是 MRSRB6 和 MRSRB7，对应 Bank6 和 Bank7，如表 4-7 所示。此寄存器 S3C2410X 只有 CL 可以设置，参照 HY57V561620T-H 手册，取 011，即 3CLKs。

表 4-7　MRSRn 寄存器定义

MSR	位	描述	初始状态
保留	[11:10]	未用	—
WBL	[9]	猝发写的长度。0：猝发(固定的)；1：保留	X
TM	[8:7]	测试模式。00：模式寄存器组；01，10 和 11：保留	XX
CL	[6:4]	$\overline{\text{CAS}}$延迟。000: 1 时钟；010: 2 时钟；011: 3 时钟；其它保留	XXX
BT	[3]	猝发类型。0：连续的；1：保留的	X
BL	[2:0]	猝发长度。000：1；其他保留	XXX

注：当代码在 SDRAM 中运行时，绝不能够重新配置 MRSR 寄存器。

4.4　I/O 系统

4.4.1　I/O 系统基础

1．I/O 接口的作用

I/O 接口，即输入输出接口，是微控制器同外界进行交互的重要通道。在主机和外围设备之间的信息交换中起着桥梁和纽带作用。

设置接口电路的必要性：

1) 解决主机 CPU 和外围设备之间的时序配合和通信联络问题；

2) 解决 CPU 和外围设备之间的数据格式转换和匹配问题；

3) 解决 CPU 的负载能力和外围设备端口选择问题。

2. I/O 接口的编址方式

对 I/O 的管理是依据 I/O 地址进行的，I/O 编址方式通常有独立编址和统一编址两种方式。

1) I/O 接口独立编址：这种编址方式是将存储器地址空间和 I/O 接口地址空间分开设置，互不影响。设有专门的输入指令(IN)和输出指令(OUT)来完成 I/O 操作。

2) I/O 接口与存储器统一编址方式：这种编址方式不区分存储器地址空间和 I/O 接口地址空间，把所有的 I/O 接口都当做是存储器的一个单元对待，每个接口芯片都安排一个或几个与存储器统一编号的地址号。不必设专门的输入/输出指令，所有传送和访问存储器的指令都可用到 I/O 接口操作。

两种编址方式有各自的优缺点，独立编址方式的主要优点是内存地址空间与 I/O 接口地址空间分开，互不影响，译码电路较简单，并设有专门的 I/O 指令，所以编程序易于区分，且执行时间短、快速性好。其缺点是只用 I/O 指令访问 I/O 端口，功能有限且要采用专用 I/O 周期和专用 I/O 控制线，使微处理器复杂化。统一编址方式的主要优点是访问内存的指令都可用于 I/O 操作，数据处理功能强；同时 I/O 接口可与存储器部分共用译码和控制电路。其缺点一是 I/O 接口要占用存储器地址空间的一部分；二是因不用专门的 I/O 指令，程序员较难区分 I/O 操作。

3. GPIO 的原理和结构

GPIO(General Purpose I/O，通用 I/O)是 I/O 的最基本形式。它是一组输入引脚或输出引脚，CPU 对它们能够进行存取。有些 GPIO 引脚能加以编程而改变工作方向。GPIO 的另一传统术语称为并行 I/O(parallel I/O)。GPIO 引脚可以被配置为多种工作模式，其中，有三种比较常用：高阻输进、推挽输出、开漏输出。

(1) 高阻输入

为减少信息传输线的数目，大多数系统中的信息传输线采用总线形式。在计算机中一般有三组总线，即数据总线、地址总线和控制总线。为防止信息相互干扰，要求连接到总线上的寄存器或存储器等，输入/输出端不仅能呈现 0、1 两个信息状态，而且还应能呈现第三个状态——高阻抗状态，此时，它们的输出似乎被开关断开，对总线状态不起作用，总线可由其他器件占用。三态缓冲器即可实现上述功能，它除具有输入/输出端之外，还有一个控制端。在不执行读操纵时，外部引脚与内部总线之间是隔离的。

(2) 推挽输出

推挽输出原理：在功率放大器电路中大量采用推挽放大器电路，这种电路中用两个晶体管构成一级放大器电路，两个晶体管分别放大输进信号的正半周和负半周。推挽放大器电路中，一个晶体管工作在导通、放大状态时，另一个晶体管处于截止状态，当输入信号变化到另一个半周后，原来导通、放大的晶体管进入截止状态，而原来截止的晶体管进入导通、放大状态，两只晶体管在不断地交替导通放大和截止变化，所以称为推挽放大器。

如图 4.13 所示为 GPIO 引脚在推挽输出模式下的等效结构示意图。锁存器执行 GPIO 引脚写操作时，在写脉冲(Write Pulse)的作用下，数据被锁存到 Q 和 $\overline{Q}$。两个 CMOS 管构成反相器，任意一个导通时都表现出较低的阻抗，但两个不会同时导通或同时封闭，最后形成的是推挽输出。在推挽输出模式下，GPIO 还具有回读功能，实现回读功能的是一个简单的三态门。

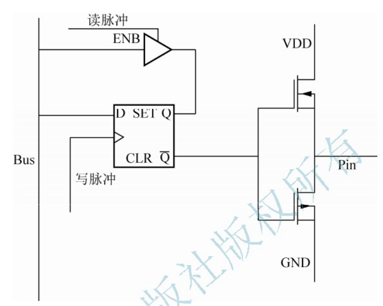


图 4.13 推挽输出电路图

(3) 开漏输出

图 4.14 为 GPIO 引脚在开漏输出模式下的等效结构示意图。开漏输出和推挽输出相比，结构基本相同，但只有下拉晶体管而没有上拉晶体管。开漏输出的实际作用就是一个开关，输出“1”时断开、输出“0”时连接到 GND(有一定内阻)。回读功能：读到的还是输出锁存器的状态，而不是外部引脚 Pin 的状态。因此开漏输出模式是不能用来输入的。

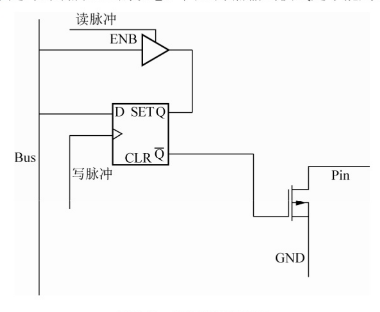


图 4.14 开漏输出电路图



开漏输出结构没有内部上拉，因此在实际应用时通常都要外接合适的上拉电阻(通常采用 4.7～10kΩ)。开漏输出能够方便地实现“线与”逻辑功能，即多个开漏的引脚可以直接并在一起(不需要缓冲隔离)使用，外接一个合适的上拉电阻，就自然形成“逻辑与”关系。开漏输出的另一种用途是能够方便地实现不同逻辑电平之间的转换(如 3.3～5V 之间)，只需外接一个上拉电阻，而不需要额外的转换电路。典型的应用例子就是基于开漏电气连接的 I^2C 总线。

4.4.2　S3C2410X 的 I/O 端口

S3C2410X 有 117 个多功能的输入输出引脚，这些端口如下。

- 端口 A(GPA)：23 个输出口；
- 端口 B(GPB)：11 个输入/输出口；
- 端口 C(GPC)：16 个输入/输出口；
- 端口 D(GPD)：16 个输入/输出口；
- 端口 E(GPE)：16 个输入/输出口；
- 端口 F(GPF)：8 个输入/输出口；
- 端口 G(GPG)：16 个输入/输出口；
- 端口 H(GPH)：11 个输入/输出口。

每个端口可以根据系统配置和设计需求通过软件配置成相应的功能。在启动主程序之前，必须定义好每个引脚的功能。如果某个引脚不用作复用功能，则可以将它配置成 I/O 脚。

1. 端口控制寄存器(GPACON-BGHCON)

在 S3C2410X 中，大部分端口都是复用的，因此需要决定每个引脚使用哪个功能。端口控制寄存器 PnCON 决定每个引脚的功能。如果 GPF0－GPF7 和 GPG0－GPG7 用于掉电模式的唤醒信号，这些端口必须被配置成中断模式。

2. 端口数据寄存器(GPADAT-GPHDAT)

如果端口被配置成输出端口，可以向 PnDAT 中的相关位写入数据；如果端口被配置成输入端口，可以从 PnDAT 中的相关位读入数据。

3. 端口上拉电阻寄存器(GPBUP-GPHUP)

端口上拉电阻寄存器控制每个端口组的上拉电阻的使能和禁止。当相关位为 0，上拉电阻使能；当相关位为 1，上拉电阻禁止；当端口上拉电阻寄存器使能时，不管引脚选择什么功能(输入、输出、数据、外部中断等)，上拉电阻都工作。

4. 外部中断控制寄存器(EXTINTN)

24 个外部中断可响应各种信号请求方式。EXTINTN 寄存器可以配置如下信号请求方式：低电平触发、高电平触发、上升沿触发、下降沿触发、双边沿触发。这 8 个外部中断引脚具有数字滤波器。只有 16 个外部中断引脚(EINT [15:0])被用于唤醒源。

5. 掉电模式和 I/O 口

所有的 GPIO 寄存器的值在掉电模式下被保存。这在时钟功率管理模块中的掉电模式下提到。EINTMASK 不能禁止从掉电模式唤醒，但是如果 EINTMASK 屏蔽了 EINT[15:4]中的 1 位，系统可以被唤醒，但是 SRCPND 中的 EINT4_7 位和 EINT8_23 位不会在唤醒后置 1。

4.5 人机交互系统

4.5.1 LCD 接口

1. LCD 的特点及分类

LCD 作为电子信息产品的主要显示器件，相对于其他类型的显示部件来说，有其自身的特点，概括如下。

(1) 低电压微功耗

LCD 的工作电压一般为 3～5V，每平方厘米的液晶显示屏的工作电流为 μA 级，所以液晶显示器件为电池供电的电子设备的首选显示器件。

(2) 平板型结构

LCD 的基本结构是由两片玻璃组成的很薄的盒子。这种结构具有使用方便、生产工艺简单等优点。特别是在生产上，适宜采用集成化生产工艺，通过自动生产流水线可以快速、大批量地生产。

(3) 使用寿命长

LCD 器件本身几乎没有劣化问题。若能注意器件防潮、防压、防止划伤、防止紫外线照射、防静电等，同时注意使用温度，LCD 可以使用很长时间。

(4) 被动显示

对 LCD 来说，环境光线越强显示内容越清晰。人眼所感受的外部信息，90%以上是外部物体对光的反射，而不是物体本身发光，所以被动显示更适合人的视觉习惯，更不容易引起疲劳。这在信息量大、显示密度高、观看时间长的场合显得更重要。

(5) 显示信息量大且易于彩色化

LCD 与 CRT 相比，由于 LCD 没有荫罩限制，像素可以做得很小，这对于高清晰电视是一种理想的选择方案。同时，液晶易于彩色化，方法也很多。特别是液晶的彩色可以做得更逼真。

(6) 无电磁辐射

CRT 工作时，不仅会产生 X 射线，还会产生其他电磁辐射，影响环境。LCD 则不会有这类问题。

液晶显示器件分类方法有多种，这里简要介绍以下几种分类方法。

1) 按电光效应分类。所谓电光效应是指在电的作用下，液晶分子的初始排列改变为其他排列形式，从而地液晶盒的光学性质发生变化，也就是说，以电通过液晶分子对光进行



了调制。不同的电光效应可以制成不同类型的显示器件。按电光效应分类，LCD 可分为电场效应类、电流效应类、电热写入效应类和热效应类。其中电场效应类又可分为扭曲向列效应(TN)类、宾主效应(GH)类和超扭曲效应(STN)类等。MCU 系统中应用较广泛的是 TN 型和 STN 型液晶器件，由于 STN 型液晶器件具有视角宽、对比度好等优点，几乎所有 32 路以上的点阵 LCD 都采用了 STN 效应结构。STN 型正逐步代替 TN 型而成为主流。

2) 按显示内容分类。LCD 可分为字段型(或称为笔画型)、点阵字符型、点阵图形型三种。字段型 LCD 是指以长条笔画状显示像素组成的液晶显示器。字段型 LCD 以七段显示最常用，也包括为专用液晶显示器设计的固定图形及少量汉字。字段型 LCD 主要应用于数字仪表、计算器中。

点阵字符型 LCD 是指显示的基本单元由一定数量的点阵组成，专门用于显示数字、字母、常用图形符号及少量自定义符号或汉字。这类显示器把 LCD 控制器、点阵驱动器、字符存储器等全做在一块印刷电路板上，构成便于应用的液晶显示模块。点阵字符型液晶模块在国际上已经规范化，有统一的引脚与编程结构。点阵字符型液晶显示模块有内置 192 个字符，另外用户可自定义 5×7 点阵字符或 5×11 点阵字符若干个。显示行数一般为 1 行、2 行、4 行三种。每行可显示 8 个、16 个、20 个、24 个、32 个、40 个字符不等。

点阵图形除了可以显示字符外，还可以显示各种图形信息、汉字等，显示自由度大。常见的模块点阵从 80×32 到 610×480 不等。

3) 按 LCD 的采光方式分类。LCD 器件按其采光方式分类，分为带背光源与不带背光源两大类。不带背光的 LCD 显示是靠背面的反射模将射入的自然光从下面反射出来完成的。大部分计数、计时、仪表、计算器等计量显示部件都是用自然光源，可以选择不带背光的 LCD 器件。如果产品需要在弱光或黑暗条件下使用可以选择带背光型 LCD，但背光源增加了功耗。

2. LCD 的控制方法

早期单片机系统集成度比较低，可扩展接口少，LCD 往往是通过 LCD 控制器连在单片机总线上，或者通过并口、串口和单片机相连。现在很多厂商都在 SoC 中集成了 LCD 控制器，使开发人员能够方便地控制 LCD。早期低端的芯片提供的一般都是 TN 型的 LCD 控制器，目前已经有越来越多的芯片提供对 TFT 型显示器的支持。

处理器内核是整个片上系统的核心，例如，ARM 的内核、MIPS 的内核等。系统总线是指处理内部的总线，例如，ARM 的 AMBA 总线，其他片上系统的外设都通过总线和处理器连接。LCD 控制器工作时，通过 DMA 请求占用系统总线，直接通过 SDRAM 控制器读取 SDRAM 中指定地址(显示缓冲区)的数据。此数据经过 LCD 控制器转换成液晶屏扫描数据的格式，直接驱动液晶屏显示。

目前市面上出售的 LCD 模块有两种类型：一种是带有驱动电路的 LCD 显示模块，这种 LCD 可以方便地与各种低端单片机进行接口，如 8051 系列单片机，但是由于硬件驱动电路的存在，体积比较大。这种模式常常使用总线方式来驱动。另一种是 LCD 显示屏，没有驱动电路，需要与驱动电路配合使用。特点是体积小，但却需要另外的驱动芯片。也可以使用带有 LCD 驱动能力的高端微处理器驱动，如 S3C2410X 微处理器。

(1) 总线驱动方式

一般带有驱动模块的 LCD 显示屏使用这种驱动方式，由于 LCD 已经带有驱动硬件电路，因此模块给出的是总线接口，便于与单片机的总线进行接口。驱动模块具有 8 位数据总线，外加一些电源接口和控制信号，而且自带显示缓存，只需要将要显示的内容送到显示缓存中就可以实现内容的显示。由于只有 8 条数据线，因此常通过引脚信号来实现地址与数据线复用，以达到把相应数据送到相应显示缓存的目的。

(2) 控制器扫描方式

以 S3C2410X 微处理器为例，S3C2410X 具有内置的 LCD 控制器，它具有将显示缓存(在系统存储器中)中的 LCD 图像数据传输到外部 LCD 驱动电路的逻辑功能。支持 DSTN(被动矩阵或叫无源矩阵)和 TFT(主动矩阵或叫有源矩阵)两种 LCD 屏，并支持黑白和彩色显示。

在灰度 LCD 上，使用基于时间的抖动算法(Time-Based Dithering Aigorithm，TBDA)和帧率控制(Frame Rate Control，FRC)方法，可以支持单色、2 级、4 级和 8 级灰度模式的灰度 LCD。在彩色 LCD 上，可以支持 16777216 色。有 7 路 DMA 通道，可支持两个 LCD 屏。列于不同尺寸的 LCD，具有不同数量的垂直和水平像素、数据接口的数据宽度、接口时间及刷新率，而 LCD 控制器可以进行编程控制相应的寄存器值，以适应不同的 LCD 显示板。

3. S3C2410X 的 LCD 控制器接口

这里以 S3C2410X 微处理器与 LQ080V3DG01 液晶屏的连接为例介绍 LCD 控制器接口。S3C2410X 内部自带一个 LCD 驱动控制器，其接口可以与单色、灰度、彩色 STN 型和彩色 TFT 型的 LCD 直接相连。但需要根据所连接的 LCD 的类型设置相应寄存器中的显示模式。LQ080V3DG01 液晶屏要求其电源电压 VDD 典型值为 3.3V/5V，并且 LCD 数据和控制信号的高电平输入电压 V_{ih} 在 2.3～5.5V 范围内，低电平输入电压 V_{il} 在−0.3～+0.9V 范围内，故可以直接与 S3C2410X 相连，其电路图如图 4.15 所示。

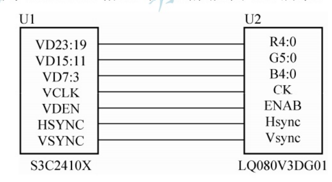


图 4.15　S3C2410 液晶屏连接电路图

S3C2410X 处理器中的 LCD 控制器内含寄存器 LCDCON1～LCDCON5。对于 LQ080V3DG01，这些寄存器具体设置如下。

1) 在 LCDCON1 中，CLKVAL 是时钟参数，对于 LQ080V3DG01，该域取 1。PNRMODE 是显示模式参数，该域取 3，表示所用模块式 TFT 型。BPPMODE 是每个像素的数据位数参数，对于 LQ080V3DG01 模块，设计时可设置成 16bpp，所以此域值取 12。



2) 在 LCDCON2 中，VBPD 对应于 LQ080V3DG01 时间参数中(表 4-8)的 H 参数，该域取 32。LINEVAL 对应 I 参数，该域值取 479。VFPD 对应于表 1 中的 G 参数，该域值取 1。

3) 在 LCDCON3 中，HBPD 对应于 LQ080V3DG01 的 C 时间参数，该域取 47。HOZVAL 对应于 D 参数，该域取 639。HFPD 对应于 E 参数，该域值取 15。

4) 在 LCDCON4 中只需要设置 HSPW 即可，它对应于 B 参数，该域值取 95。

5) 在 LCDCON5 中，BPP24BL 用于决定 24bpp 视频存储器的大小端模式，该域值取 0。FRM565 决定 16bpp 视频输出数据的格式。

表 4-8 LQ080V3DG01 的时间参数

符号	A	B	C	D	E	F	G	H	I	J
数值	800	96	48	640	16	525	2	33	480	10

4.5.2 触摸屏的应用

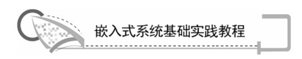
1. *触摸屏介绍*

触摸屏按其工作原理的不同分为表面声波屏、电容屏、电阻屏和红外屏等。而其中电阻屏触摸屏最为常用。

电阻触摸屏的主要部分是一块与显示器表面非常配合的电阻薄膜屏，这是一种多层的复合薄膜，它以一层玻璃或硬塑料平板作为基层，表面涂有一层透明氧化金属(ITO 氧化铟，透明的导电电阻)导电层，上面再盖有一层外表面硬化处理、光滑防擦的塑料层，它的内表面也涂有一层 ITO 涂层，它们之间有许多细小的(小于 1/1000in)透明隔离点把两层导电层隔开绝缘。当手指触摸屏幕时，两层导电层在触摸点位置就有了接触，控制器侦测到这一接触并计算出(X，Y)的位置，再根据模拟鼠标的方式运作。这就是电阻技术触摸屏的最基本的原理。例如，使用电阻屏的 Nokia 5800 手机可以在-15℃～+45℃的温度下正常工作，体现出了电阻屏的一些优势。

表面声波技术是利用声波在物体的表面进行传输，当有物体触摸到表面时，阻碍声波的传输，换能器侦测到这个变化，反映给计算机，进而进行鼠标的模拟。表面声波屏特点：清晰度较高，透光率好；高度耐久，抗刮伤性良好；一次校正不漂移；反应灵敏；适合于办公室、机关单位及环境比较清洁的场所。典型的应用案例有自助服务设备、零售终端/POS 机、教育培训、游戏机、工业控制、ATM 机、医疗设备等，第二类是新兴的触控计算机设备领域，包括触控笔记本式计算机、触控平板电脑、触控一体化计算机和触控桌面显示器。10～32in(in=2.54 厘米)的固定式设备触摸屏的其他应用还有很多，如电视台使用的直播室大屏幕显示器、军事作战指挥系统、城市应急管理系统等。

电容屏利用人体的电流感应进行工作。用户触摸屏幕时，由于人体电场，用户和触摸屏表面形成以一个耦合电容，对于高频电流来说，电容是直接导体，于是手指从接触点吸走一个很小的电流。这个电流分从触摸屏的四角上的电极中流出，并且流经这四个电极的



电流与手指到四角的距离成正比，控制器通过对这四个电流比例的精确计算，得出触摸点的位置。代表产品就是苹果 iPad touch 和 iPad 系列产品。

2. 电阻屏的工作原理及控制方法

电阻式触摸屏一般由三部分组成，两层透明的阻性导体层及这两层之间的隔离层。在没有外力时两个阻性导体层中间被微小透明的绝缘“分隔点”隔开，两层没有电气联系。如果触摸屏的某一点被外力作用，则在这点的上下两阻性导体层便会相互接触。

工作时触摸屏上下导体层的电阻网络是交替工作的，当其中一层两端加上电压在层中的阻性导体中形成均匀的电压梯度时，另一层就作为侦测层工作。很显然，由于接触点的位置不同，侦测层所得到的是一个与位置有关的电压，并且侦测层不应该对这个电压产生什么影响。工作时触摸屏的引脚状态如图 4.16 所示。将接触点的电压通过 AD 转换读入 CPU 中，再经过一定的运算处理，就可以得到触摸点的坐标。为了能分别检测一个点的 *X* 坐标和 *Y* 坐标，一阻性层形成的电压梯度应在 *X* 方向上，另一阻性层的电压梯度应在 *Y* 方向上。因此，要获取屏上触摸点的坐标首先要对触摸屏的引脚进行切换控制，使其处于合适的状态，然后通过 ADC 转换采集接触点处的电压值，最后对采集的电压进行平均、坐标变换等后续处理，得到触摸点的坐标。

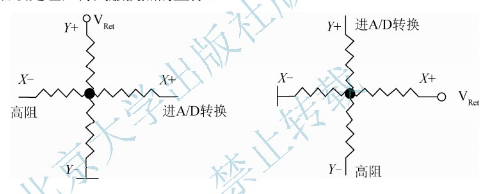


图 4.16　电阻屏原理示意图

3. S3C2410X 的触摸屏控制接口

S3C2410X 的触摸屏控制接口是与 ADC 共用的，它内部集成了一个 10 位的 ADC，有 8 路模拟输入通道。其中，通道 AIN 作为触摸屏接口的 *X* 坐标输入，通道 AIN[5]作为触摸屏接口的 *Y* 坐标输入。在 2.5MHz 的 A/D 转换时钟频率下它的最快转换速度为 500kSPS。同时，S3C2410X 也提供了触摸屏引脚的控制信号 YMON、nXPON、XMON、nYPON，通过外部晶体管对触摸屏引脚 *X*+、*X*−、*Y*+、*Y*−进行切换控制。

YMON、nXPON、XMON、nYPON 等信号是由 ADCTSC(ADC Touch Screen Control register, ADC 触摸屏控制寄存器)中有关位来控制的。根据这些位的定义和触摸屏的控制要求，触摸屏接口电路设计如图 4.17 所示，其中外部晶体管采用双 MOS 管 FDC6321，电源电压要求为 3.3V。图 4.17 中的 RC 滤波电路可以过滤传递给 S3C2410X 模数转换输入接口信号中的干扰，以利于后续的软件处理。

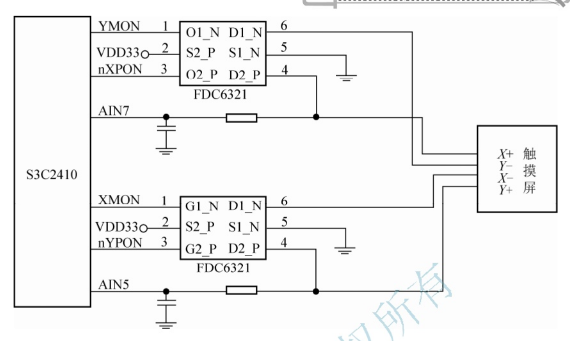


图 4.17　S3C2410 与触摸屏的连接

4.5.3　键盘接口

1．键盘模型及接口

键盘是有若干个按键构成的开关矩阵，它是嵌入式系统中最简单的数字量输入设备，操作员通过键盘输入数据或命令，实现简单的人-机通信。

(1) 键盘模型

键盘的基本电路是一个接触开关，通、断两种状态分别表示 0 和 1，微处理器可以很容易地检测到开关的闭合。当开关打开时，提供逻辑“1”；当开关闭合时，提供逻辑“0”。

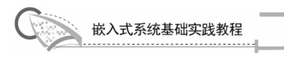
(2) 键盘接口

键盘接口按照不同的标准有不同的分类方法。按键盘排布的方式可分成独立方式和矩阵方式；按读入键值的方式可分为直读方式和扫描方式；按是否进行硬件编码可分成非编码方式和硬件编码方式；按微处理器响应方式可分为中断方式和查询方式。将以上各种方式组合可构成不同的键盘接口方式。以下介绍较为常用的两种方式。

1) 独立方式是指将每个独立按键一对一的方式直接接到 I/O 输入线上。读键值时直接读 I/O 口，每一个键的状态通过读入键值来反映，所以也称这种方式为一维直读方式，按习惯称为独立式。这种方式查键实现简单，但占用 I/O 资源较多，一般在键的数量较少时采用。

2) 矩阵方式是用 n 条 I/O 线组成行输入口，m 条 I/O 线组成列输出口，在行列线的每个交点上设置一个按键。读键值方法一般采用扫描方式，即输出口按位轮换输出低电平，再从输出口读入键信息，最后获得键码。这种方式占用 I/O 线较少，在实际应用系统中采用较多。

设计键盘的时候，通常小于四个键盘的应用，可以使用独立式接口。如果多于四个按键，为了减少微处理器的 I/O 端口线的占用，可以使用矩阵式键盘。



2. 键盘的基本问题

为了能实现对键盘的编程至少应该了解下面几个问题：第一，如何识别键盘上的按键？第二，如何区分按键是被真正按下，还是抖动？第三，如何处理重键问题？了解这个问题有助于键盘编程。

(1) 键盘的识别

如何知道键盘上哪个键按下就是键盘的识别问题。若键盘上闭合键的识别由专用硬件实现，称为编码键盘；而靠软件实现的称为未编码键盘。在这里主要讨论未编码键盘的接口技术。识别是否有键被按下，主要有查询法、定时扫描法与中断法等。而要识别键盘上哪个键被按下主要有行扫描法和行反转法。

(2) 抖动问题

当手按下一个键时，会出现所按的键在闭合位置和断开位置之间跳几下才稳定到闭合状态的情况，当释放一个按键时也会出现类似的情况，这是抖动问题。抖动持续的时间因操作而异，一般为5～10ms，稳定闭合时间一般为十分之几秒，由操作者的按键动作所确定。在软件上，解决抖动的方法通常是延迟等待抖动的消失或多次识别判定。

(3) 重键问题

所谓重键问题就是有两个及两个以上按键同时处于闭合状态的处理问题。在软件上，处理重键问题通常有连锁法和巡回法。

3. S3C2410X的键盘接口实例

36个按键按6×6方式排列，如图4.18所示，其中行线分别接S3C2410X的GPB0、GPB1、GPB2、GPB3、GPB4、GPB5 口，列线分别接GPF0、GPF1、GPG3、GPG5、GPG6、GPG7 口。列线可以复用EINT 0、EINT 1、EINT 11、EINT 13、EINT 14、EINT 15 口，外接上拉电阻保证按键在未按下时中断口处于稳定的高电平状态。

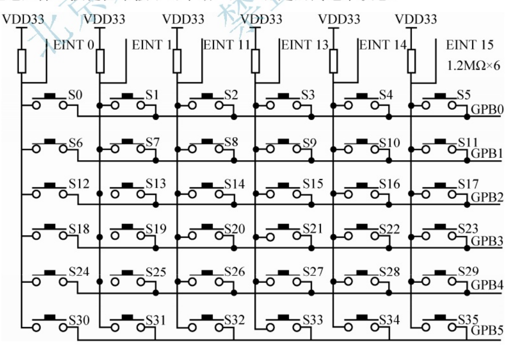


图4.18　S3C2410X与键盘的连接

4.6 调 试 接 口

4.6.1 JTAG 逻辑结构

开发 JTAG 标准的主要用途是为了对 PCB 板上的芯片进行芯片功能测试和与其他芯片的互连接性测试。图 4.19 是 JTAG 测试逻辑的示意图。JTAG 测试逻辑结构中应包括四部分：测试访问口(Test Access Port , TAP)、TAP 控制器、指令寄存器以及一组测试数据寄存器。

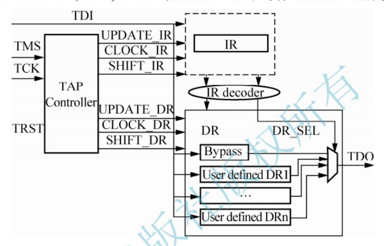


图 4.19　JTAG 逻辑结构示意图

测试访问口包括四个必选信号：TCK(测试时钟)、TMS(测试模式选择)、TDI(测试数据输入)和 TDO(测试数据输出)。另外，测试访问口还包括一个可选信号 TRST(测试复位)。TAP 控制器实现了一个具有 16 状态的状态机，由 TMS 信号控制状态机的状态转移。IR(指令寄存器)和 DR(数据寄存器)都分别由移位级和锁存级两级构成。TAP 状态机可以分别选中 IR 或 DR 进行操作。在 Capture 状态下，IR(或 DR)锁存级寄存器的内容被捕获到移位级。在 Shift 状态下，TDI 信号上的数据被串行移入 IR(或 DR)寄存器的移位级，同时移位级中的内容通过 TDO 信号串行移出。

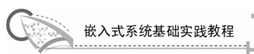
状态机会保持多拍的 Shift 状态，直到所需的数据被移入或移出移位级。在 Update 状态下，Shift 状态下串行移入移位级的内容被一次性更新到锁存级。移位级与外部通信，锁存级产生芯片内部逻辑所需的控制或时序信号。通过控制 TAP 状态机在几个状态间转换，就可以对芯片内部的模块进行测试。

4.6.2 JTAG 状态和工作过程

1. JTAG 状态

TAP 有 16 个状态，如图 4.20 所示，状态的转换由 TMS 控制。

- Test-Logic Reset：系统上电后，TAP Controller 自动进入该状态。在该状态下，测试部分的逻辑电路全部被禁用。
- Run-Test/Idle：在不同操作间的一个中间状态。这个状态下的动作取决于当前指令寄存器中的指令。



Test-Logic/Reset
Run-Test/Idle
Select-DR-Scan
Select-IR-Scan
Captuer-DR
Capture-IR
Shift-DR
Shift-IR
Exit1-DR
Exit1-IR
Pause-DR
Pause-IR
Exit2-DR
Exit2-IR
Update-DR
Update-IR
TMS=0
TMS=1

图 4.20　TAP 状态机



- Select-DR-Scan：临时的中间状态。
- Capture-DR：在 TCK 上升沿，芯片输出引脚上的信号将被“捕获”到与之对应的数据寄存器的各个单元中去。
- Shift-DR：由 TCK 驱动，每一个时钟周期，被连接在 TDI 和 TDO 之间的数据寄存器将从 TDI 接收一位数据，同时通过 TDO 输出一位数据。
- Update-DR：由 TCK 上升沿驱动，数据寄存器当中的数据将被加载到相应的芯片引脚上去，用以驱动芯片。
- Select-IR-Scan：临时的中间状态。
- Capture-IR：一个特定的逻辑序列装载到指令寄存器中。
- Shift-IR：与 Shift-DR 类似，对应指令寄存器。
- Update-IR：新指令将被用来更新指令寄存器。

2. JTAG 工作过程

1) JTAG 处于挂起状态，JTAG 的扫描单元并不影响设备信号的输入输出。

2) 在 JTAG 状态机的 Capture-DR 状态，把 I/O 口上的数据捕获到 JTAG 扫描单元的移位寄存器上。

3) 在 JTAG 状态机的 Shift-DR 状态，TCK 的一次跳变，把数据从 TDI 移位到 JTAG 移位寄存器的高位上，并从 TDO 输出移位寄存器的低位。

4) 经过六个 TCK 的时钟可以把整个捕获到的 JTAG 链的移位寄存器上的数据移出，并且，把新的数据移入 JTAG 链。

5) 在 JTAG 状态机的 Update-DR 状态，可以把新的数据锁定到设备的输入或者输出 I/O 口上，从而完成了一次 JTAG 的数据更新。

4.7　案 例 分 析

本章导入案例中给出的是武汉创维特信息技术有限公司设计的 JXARM9-2410-1 嵌入式实验教学平台。该实验教学平台以 S3C2410 芯片为基础扩展了最小嵌入式系统，并以此最小系统为核心扩展了面向控制的各类接口，支持多种操作系统，提供了嵌入式软件开发实验的多个例程。下面利用本章所学知识对该案例做进一步分析。

4.7.1　嵌入式最小系统

JXARM9-2410-1 实验教学平台的核心模块中包含了嵌入式最小系统的微处理器、晶振电路、存储器，预留了调试接口、I/O 接口连接插座。核心模块主要包含以下几方面。

1) 64MB 的 SDRAM 存储器，由两片 16 位数据宽度的 SDRAM 存储器组成，地址为 0x30000000～0x33FFFFFF。

2) 32M B NOR Flash 存储器和 8MB NAND Flash，NOR Flash 内部存放启动代码 Boot loader、Linux 内核映像、I^2S 测试声音文件等。其数据宽度为 32 位，地址为 0x00000000～0x01FFFFFF；NAND Flash 中包含一个 CRAMFS 文件系统，在 Linux 中使用。

3) 晶振电路：核心模块上有两片晶振。一片 12MHz 晶振为内部锁相环提供震荡输入，另一片 32.768kHz 晶振为 RTC 电路提供震荡输入。

4) 电源及复位：电源部分有两个电压转换模块，将 5V 和 3.3V 的电压分别转换成 3.3V 和 1.8V。核心模块基于 74HC04 设计了复位电路。

5) 调试接口：实验箱上有 20 针标准 JTAG 接口，可以用于高速仿真调试；另外还有一个建议仿真调试口，可以直接连接在计算机并口上。

6) 其他接口：为实现良好的人机交互，实验箱扩展了显示屏、触摸屏、键盘、鼠标等交互接口。

4.7.2 面向具体应用的接口

在通信方面，实验箱扩展了以太网接口、USB 接口、标准计算机打印口(并口)以及 GPRS 无线通信模块。

面向控制领域的应用，实验箱扩展了两相步进电机、RS485 总线接口、CAN 总线接口。

为提供对多种存储介质的支持，实验箱扩展了标准 IDE 硬盘接口、标准 CF 卡接口以及 SD/MMC 卡接口。

4.7.3 软件环境

实验箱的软件系统及开发平台也是二次开发应注意的要素，案例中的实验箱可稳定运行 Linux、Windows CE、VxWorks、Nucleus、μC/OS-Ⅱ等嵌入式实时操作系统，并可任意内置多操作系统。提供多达 58 项实验项目，分为嵌入式基础实验、嵌入式接口实验、嵌入式 BootLoader 实验、嵌入式操作系统(μC/OS-Ⅱ及 Linux)基础实验/接口实验/图形用户界面(GUI)实验、高级应用实验等类别，为二次开发提供良好实验基础。

本 章 小 结

本章以 ARM 核为基础，以 S3C2410X 为微处理器芯片案例，介绍了以微处理器为核心，不断扩展外围接口，形成最小硬件系统的硬件平台构建过程；分析了以 ARM9TDMI 为基础的 ARM920T 核；介绍了存储系统、I/O 系统、人机交互接口、JTAG 接口的基础知识；以 S3C2410X 为例，讲解了该芯片扩展外围系统的方法和步骤。

(1) 嵌入式最小系统：介绍构建嵌入式硬件平台的基本要素，它实际应用系统的瘦身，更是硬件设计的基础性步骤。嵌入式最小系统通常包括微处理器、存储器、电源电路、复位电路、晶振电路、串口等。

(2) S3C2410X 微处理器芯片：该芯片是前文讲述的 ARM 核及其指令集的物化与扩展。它不仅包括 ARM 核，而且有丰富的外围接口，适合于手持设备的应用。

(3) 存储系统：比较了不同存储器容量、速度等特点，介绍了分级存储结构，分析了 S3C2410X 的存储接口，列举了 S3C2410X 配置 SDRAM 存储器的案例。

(4) I/O 系统：讲解了 I/O 系统编址方法、控制策略、GPIO 特点，分析了 S3C2410X 的 GPIO 系统。



(5) 人机交互系统：着重讲解 LCD、触摸屏、键盘等基本交互接口；叙述了 LCD、触摸屏的分类与特点，以实际案例讲解 S3C2410X 的 LCD、触摸屏、键盘连接电路。

(6) JTAG接口：在嵌入式系统中，广泛采用JTAG接口作为调试接口，本章介绍了JTAG的原理与工作过程。


阅读材料

NOR Flash 和 NAND Flash

NOR Flash 和 NAND Flash 是现在市场上两种主要的非易失闪存技术。Intel 于 1988 年首先开发出 NOR Flash 技术，彻底改变了由 EPROM 和 EEPROM 一统天下的局面。紧接着，1989 年，东芝公司发表了 NAND Flash 结构，强调降低每比特的成本，更高的性能，并且像磁盘一样可以通过接口轻松升级。但是经过十多年之后，仍然有相当多的硬件工程师分不清 NOR Flash 和 NAND Flash。许多业内人士 NAND Flash 技术相对于 NOR Flash 技术的优越之处也不是很明确，因为大多数情况下闪存只是用来存储少量的代码，这时 NOR Flash 闪存更适合一些。而 NAND Flash 则是高数据存储密度的理想解决方案。NOR Flash 的特点是芯片内执行(eXecute In Place，XIP)，这样应用程序可以直接在 Flash 内运行，不必再把代码读到系统 RAM 中。NOR Flash 的传输效率很高，在 1～4MB 的小容量时具有很高的成本效益，但是很低的写入和擦除速度大大影响了它的性能。NAND Flash 结构能提供极高的单元密度，可以达到高存储密度，并且写入和擦除的速度也很快。应用 NAND Flash 的困难在于 Flash 的管理和需要特殊的系统接口。下面从几个方面比较 NOR Flash 和 NAND Flash 的异同。

(1) 性能比较

Flash 是非易失存储器，可以对存储器单元块进行擦写和再编程。任何 Flash 器件的写入操作只能在空或已擦除的单元内进行，所以大多数情况下，在进行写入操作之前必须先执行擦除。NAND Flash 器件执行擦除操作是十分简单的，而 NOR Flash 则要求在进行擦除前先要将目标块内所有的位都写为 0。由于擦除 NOR Flash 器件时是以 64～128KB 的块进行的，执行一个写入/擦除操作的时间为 5s，与此相反，擦除 NAND Flash 器件是以 8～32KB 的块进行的，执行相同的操作最多只需要 4ms。

(2) 接口差别

NOR Flash 带有 SRAM 接口，有足够的地址引脚来寻址，可以很容易地存取其内部的每一个字节。NAND Flash 器件使用复杂的 I/O 口进行串行存取数据，各个产品或厂商的方法可能各不相同。8 个引脚用来传送控制、地址和数据信息。NAND Flash 读和写操作采用 512 字节的块，类似于硬盘管理此类操作。

(3) 容量和成本

NAND Flash 的单元尺寸几乎是 NOR Flash 器件的一半，由于生产过程更为简单，NAND Flash 结构可以在给定的模具尺寸内提供更高的容量，也就相应地降低了价格。NOR Flash 占据了容量为 1～16MB 闪存市场的大部分，而 NAND Flash 只是用在 8～128MB 的产品当中，这也说明 NOR Flash 主要应用在代码存储介质中，NAND Flash 适合于数据存储，NAND Flash 在 CompactFlash、Secure Digital、PC Cards 和 MMC 存储卡市场上所占份额最大。

(4) 耐用性

在 NAND Flash 中每个块的最大擦写次数是一百万次，而 NOR Flash 的擦写次数是十万次。NAND Flash 存储器除了具有 10 比 1 的块擦除周期优势，典型的 NAND Flash 块尺寸要比 NOR Flash 器件小 8 倍，每个 NAND Flash 存储器块在给定的时间内的删除次数要少一些。

(5) 易用性

可以非常直接地使用基于 NOR Flash，可以像其他存储器那样连接，并可以在上面直接运行代码。由

于需要 I/O 接口，NAND Flash 要复杂得多。各种 NAND Flash 器件的存取方法因厂家而异。在使用 NAND Flash 器件时，必须先写入驱动程序，才能继续执行其他操作。向 NAND Flash 器件写入信息需要相当的技巧，因为设计师绝不能向坏块写入，这就意味着在 NAND Flash 器件上自始至终都必须进行虚拟映射。

(6) 软件支持

在 NOR 器件上运行代码不需要任何的软件支持，在 NAND Flash 器件上进行同样操作时，通常需要驱动程序，也就是内存技术驱动程序(MTD)，NAND Flash 和 NOR Flash 器件在进行写入和擦除操作时都需要MTD。使用 NOR Flash 器件时所需要的 MTD 要相对少一些，许多厂商都提供用于 NOR Flash 器件的更高级软件，这其中包括 M-System 的 TrueFFS 驱动，该驱动被 Wind River System、Microsoft、QNX Software System、Symbian 和 Intel 等厂商所采用。驱动还用于对 DiskOnChip 产品进行仿真和 NAND Flash 的管理，包括纠错、坏块处理和损耗平衡。

习　　题

一、选择题

1．属于 LCD 三种显示方式的是(　　)。

A．投射型、反射型、透射型

B．投射型、透反射型、透射型

C．反射型、透射型、透反射型

D．投射型、反射型、透反射型

2．JTAG 的引脚 TCK 的主要功能是(　　)。

A．测试时钟输入

B．测试数据输入，数据通过 TDI 输入 JTAG 口

C．测试数据输出，数据通过 TDO 从 JTAG 口输出

D．测试模式选择，TMS 用来设置 JTAG 口处于某种特定测试模式

二、判断题

1．JTAG 是一种嵌入式系统中常用大数据传输接口。　　(　　)

2．LCD 是一种输出设备。　　(　　)

3．CPU 和 I/O 接口之间常见的数据交换方式包括 DMA(直接内存访问)方式、查询方式、中断方式等。　　(　　)

三、问答题

1．S3C2410X 存储系统有哪些特征？

2．嵌入式最小系统中，处理器、时钟、内存、电源各起什么作用？这些和计算机系统的基本操作有什么对应关系？

3．比较 SRAM、NOR FLASH、SDRAM 和 NAND FLASH 等几种存储芯片的特点和用途？

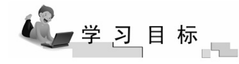
4．典型的 I/O 接口的编址方式有哪两种，各有什么特点？

5．简述 LCD 的显示原理。

6．简述 JTAG 的工作原理。

第 5 章

嵌入式 C 语言编程基础



掌握 C 语言关键字和运算符的使用；
掌握函数的使用；
理解预处理的特点并掌握使用方法；
掌握如何利用指针处理各种程序单元。

知 识 结 构

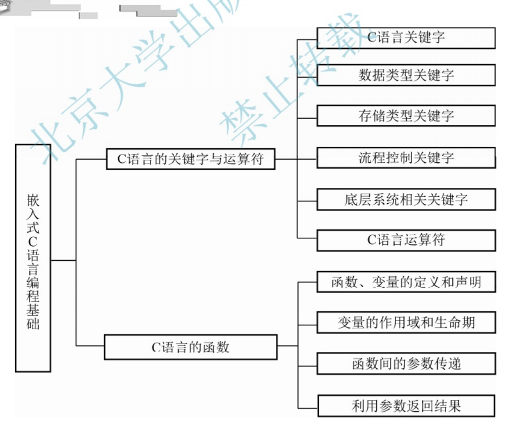


图 5.1　嵌入式 C 语言编程基础知识结构图

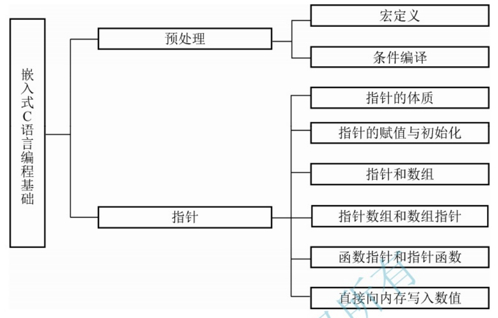


图 5.1　嵌入式 C 语言编程基础知识结构图(续)



导入案例

美国东部时间 2012 年 8 月 6 日凌晨 1 时 30 分(北京时间 6 日 13 时 30 分)，新型火星探测器“好奇”号计划着陆火星表面。作为迄今为止设计最为复杂精密的火星探测器，“好奇”号探测车采用的是风河公司业界领先的 VxWorks 实时操作系统(RTOS)。采用 VxWorks 系统，“好奇”号完成被称为 EDL(进入火星大气层、下降以及着陆)的复杂着陆过程。由于宇宙飞船安全着陆需要绝对的精确度，这一过程被称为“恐怖七分钟”。从 2011 年 11 月 26 日火箭离开地球那一刻起一直到任务完成，VxWorks 作为火星探测车的核心操作系统，将在本次具有历史意义的活动上发挥至关重要的作用。

整个火星车里有 250 万行 C 程序代码，运行在 VxWorks 操作系统上，精准无误，以毫微妙计算。这些代码运行在 BAE 制造的 RAD750 处理器上，它们包括 150 个独立模型，每个承担不同的功能。高度耦合的模块被抽象成组件，完成一个特定的功能或者行为。这些组件被进一步组合为层，整个火星登陆车包括不超过 10 个顶级层组织。



图 5.2 “好奇”号发回的火星表面照片(来自 www.nasa.gov)



代表了当今最高科技水平的“好奇”号的软件编写主要使用的是 C 语言，和人们日常生活关系密切的很多电子设备，如数码照相机、机顶盒、路由器等，它们的控制系统都极其复杂，内部功能的实现都在几十万至几百万代码。这几十万到几百万行的代码，主要是用 C 语言完成的。可见，C 语言在嵌入式软件的开发中起到了中流砥柱的作用。

C 语言具有直接读写内存的能力，并且可以直接对位进行操作，可以实现汇编语言的大部分功能。C 语言的目标代码执行效率高，仅比汇编语言程序的目标代码低 10%～20%。并且 C 语言程序具有很好的可移植性。

总之，C 语言兼具高级语言和汇编语言的双重优势，使之成为嵌入式软件开发中使用最多并占据统治地位的一种编程语言。

在嵌入式软件的开发中，目前常用的编程语言有汇编、C、C++、Java 等。汇编语言针对具体的处理器，可以操作一切硬件资源，但是编写上层程序显得烦琐。C 语言作为一种具有“低级”语言特性的高级语言，既可以用于上层应用程序的编写，又可以像汇编语言那样直接操作硬件。C++和 Java 都是面向对象的高级语言，一般用在上层应用程序的开发中。

从当前嵌入式软件开发的实际情况来看，使用最多的编程语言是 C 语言，因此熟练掌握 C 语言编程是进行嵌入式软件开发的基本要求。由于要对硬件设备和内存直接进行操作，要编写驱动程序、通信协议等底层软件，因此使用 C 语言进行嵌入式软件开发的过程中，需要对 C 语言有比较透彻深入的理解。

5.1　C 语言的关键字与运算符

5.1.1　C 语言关键字

在 C 语言中，关键字就是 C 语言本身预定义的符号，编写程序时具有特定的含义和功能，指示计算机执行特定的操作，用户自定义的标识符(常量、变量、函数名、宏等)不得与关键字冲突。ANSI C 中一共有 32 个关键字，表 5-1 对关键字的含义做一个简单的回顾。

表 5-1　C 语言的关键字

关键字	含义
auto	声明自动变量
short	声明短整型变量或函数
int	声明整形变量或函数
long	声明长整型变量或函数
float	声明浮点型变量或函数
double	声明双精度变量或函数
char	声明字符型变量或函数
struct	声明结构体变量或函数
union	声明共用数据类型

续表

关键字	含义
enum	声明枚举类型
typedef	给数据类型取别名
const	声明只读变量
unsigned	声明无符号类型变量或函数
signed	声明有符号类型变量或函数
extern	声明该量是在其他文件中定义
register	声明寄存器变量
static	声明静态变量
volatile	每次都从内存中读取变量，不进行读取优化
void	声明函数无返回值或无参数，声明无类型指针
if	条件语句
else	条件语句否定分支(与 if 连用)
switch	用于开关语句
case	开关语句分支
for	一种循环语句
do	循环语句的循环体
while	循环语句的循环条件
goto	无条件跳转语句
continue	结束当前循环，开始下一轮循环
break	跳出当前循环或分支
default	开关语句中的“其他”分支
sizeof	计算数据类型长度
return	子程序返回语句(可以带参数，也可不带参数)

表 5-1 的 32 个关键字中，按照功能，大致可以分成以下四类。

1) 数据类型关键字：char、short、int、long、float、double、signed、unsigned、void、enum、struct、union、const。

2) 存储类型关键字：auto、register、static、extern。

3) 流程控制关键字：if、else、switch、case、default、while、do、for、break、continue、return、goto。

4) 底层系统相关关键字：sizeof、typedef、volatile。

本书不再对 32 个关键字逐一介绍，而是就关键字使用中容易出错的地方，以及和嵌入式软件开发密切相关的部分进行重点介绍。

5.1.2 数据类型关键字

1. 数据类型的位长

使用 C 语言进行程序设计时，首先要明确在程序运行的处理器上，各种数据类型的长



度以及由此而决定的可以表示的数值范围。因为 ANSI C 标准并没有对简单数据类型的长度给出严格的规定，仅仅是给出了各种数据类型的长度的最小约定。表 5-2 列出各种 ANSI C 的各种整数类型的最小长度及表示范围。

表 5-2　各种整数类型的位长和取值范围

数据类型	长度(位数)	ANSI C 标准最小范围
signed char	8	-127～127
unsigned char	8	0～255
signed short	16	-32767～32767
unsigned short	16	0～65535
signed int	16	-32767～32767
unsigned int	16	0～65535
signed long	32	-2147483647～2147483647
unsigned long	32	0～4294967295

在嵌入式软件的开发中，使用的处理器和编译器往往差别较大，导致同样的数据类型在不同的平台上位长不同，表示的数据范围也不一致，如果没有意识到这一点，就可能会由于选用的数据类型不合适而导致程序运行出错。例如，(unsigned) int 这种数据类型，在 32 位的 ARM 微处理器上其长度是 32 位，而在大多数 8 位微处理器上其长度是 16 位。因此，在某种硬件平台下使用某种编译器进行嵌入式软件开发前，一定要事先阅读相关的硬件及软件厂商提供的资料，明确各种数据类型的位长及表示范围，并且最好事先编写测试程序进行验证。

另外，一个字节的长度未必总是 8 位，有时是 16 位。例如，在 TI 的 Code Composer Studio 环境下的 TMSC32x/C2xx/C5x 系列 DSP 的一个字节就是 16 位，而不是通常的 8 位。在上述情况下，因为已经规定了一个字节是 16 位，所以当用 sizeof(char)计算 char 数据类型所占空间时，结果仍然是 1，但是 char 所能表示的数值范围却大大扩展了。实际上，TMSC32x/C2xx/C5x 系列 DSP 在 CCS 编译器下，char、short、int 三种数据类型的位长是一样的，都占用 16 位的存储空间。

2. 字节存储顺序

多于一个字节的数据在内存中存放时，有大端存储和小端存储两种存放顺序。如果数据的低字节在内存高地址处，高字节在内存低地址处，称为大端存储；如果数据的低字节在内存低地址处，高字节在内存高地址处，称为小端存储。假设有一个 32 位的 int 型整数 0x12345678，它在内存中的大端存储和小端存储分别如图 5.3 所示。

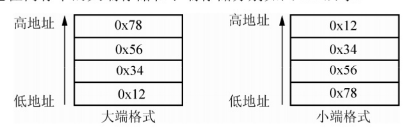


图 5.3　大端存储和小端存储

微处理器使用哪种存储格式并无好坏之分，技术的实现上也不存在难易差别，具体采用哪种格式一般取决于习惯。目前 Intel 公司的 x86 系列 CPU 均采用小端存储，IBM、Freescale 公司的处理器大多采用大端存储，ARM 处理器既支持大端存储，也支持小端存储。

如果仅仅是编写运行在单机上的上层应用程序，程序员基本不用考虑大端、小端存储的问题，但是如果编写涉及底层硬件操作或者通信协议的程序时，就必须考虑大小端的问题。因此，读者应该掌握通过编写程序测试当前机器的存储格式的方法。下面给出一段测试大小端的程序片段。

```
int check_endian()
{
    union check
    {
        int i;
        char ch;
    }u;
    u.i = 1;
    return ( u.ch == 1 );
}
```

3．浮点数与 0 值的比较

在目前的 C 计算机中，对于浮点数的存储普遍采用 IEEE-754 标准，即浮点数在内存中实际上是用科学计数法以指数的形式存储的。浮点数在内存中的存储如表 5-3 所示。

表 5-3　浮点数的存储格式

浮点数	第 31 位	第 30～23 位	第 22～0 位
float	符号位	阶码	尾数
	第 63 位	第 62～52 位	第 51～0 位
double	符号位	阶码	尾数

注：符号位中，0 表示正，1 表示负。

在本书中，关于浮点数的存储方法不过多涉及，感兴趣的读者可参考计算机组成与结构方面的书籍。但是读者应该清楚地知道，按照 IEEE 的标准存储浮点数时，由于 10 进制的小数在转化成 2 进制时，一是因为“乘 2 取整”可能不会穷尽，二则受限于浮点数尾数的位数，浮点数在内存中的表示和其绝对值存在误差是一种正常现象。因此浮点数之间互相比较时不能简单的采用“= =”或者“！ =”来进行，而应该采用判断两数相减之差是否在规定的区间范围内的方法进行。假设有两个浮点数 x 和 y，如欲比较它们的大小，正确的代码应该如下所示：

```
#define EPINSON  (1e-6)
//如果下面的条件成立,则可以认为 x 和 y 相等
```



```
        if( (x - y ) < EPINSON || ( y - x ) < EPINSON )
        {
            do_sth();
        }
```

5.1.3　存储类型关键字

1．auto 关键字

在一个函数内部定义的变量称为内部变量或者局部变量，局部变量的存储类型默认是 auto 型，一般情况下 auto 关键字可以省略不写。具有 auto 存储类型的变量的作用域是它所处的函数或者复合语句，生命期是其所在的函数被调用期间。

auto 变量存储在程序的栈区之中，在嵌入式系统特别是一些单片机系统中，由于系统的内存空间很小，程序的栈空间也就相应的很小，有时只有几十个字节。在这种情况下，使用 auto 变量时一定要防止栈的溢出。栈发生溢出时，程序的运行会出现一些莫名其妙的错误，但是从代码上来看程序又是正确的。发生这种情况时，就要考虑栈是否发生了溢出。

2．static 关键字

static 字面的意思是静态的，在 C 语言中，static 可以用来修饰变量或者函数。

(1) 修饰局部变量

auto 型局部变量的生命期仅在定义它的函数运行期间，当它所在的函数运行结束之后，局部变量占有的栈空间被系统回收，局部变量也就不存在了。当用 static 修饰局部变量时，局部变量的生命期是程序运行的整个生命期，当 static 局部变量所处的函数运行结束之后，static 变量依然存在，其保存的值也不会变化，但是其他的函数却无法再使用这个局部变量，因此，static 局部变量具有局部的可见性和全局的生命期。

(2) 修饰全局变量

一般情况下，全局变量属于外部变量，不但在定义该变量的文件中可用，在其他文件中通过 extern 关键字也可以引用。但是如果一个软件系统由多人开发，在不同的文件中需要使用同名的外部变量时，那么在该外部变量的前面可以加上 static，这样这个外部变量就变成仅在本文件中可见，在其他文件中不可见，即使加上 extern 修饰也不行。

(3) 修饰函数

当用 static 修饰函数时，其作用和 static 修饰全局变量一样：被 static 修饰的函数仅在定义它的文件中可见，其他文件中不可见。

3．extern 关键字

在 C 语言中，默认情况下外部变量和函数仅在定义它们的文件中才能访问，其他文件中不能访问这些外部变量和函数。为了扩大外部变量和函数的使用范围，可以在需要使用其他文件中定义的外部变量的文件中，对其他文件中定义的外部变量或函数，前面加上 extern 关键字，再做一次说明。在一个文件中使用其他文件中定义的外部变量和函数，可以有两种方式：一是在*.c 文件中对外部变量用 ertern 进行说明；二是把外部变量和函数的

说明放在一个*.h 文件中，在需要使用这些外部变量和函数的文件中包含相应的*.h 文件。

通过对外部变量用 extern 进行说明，扩展了外部变量和函数的使用范围，不同的文件之间可以使用外部变量进行通信。

在一个文件中定义外部变量时，定义出现在所有函数的外部，且前面不加存储类型关键字。形式如下：

```
int a,b;
void f(void)
{
    //函数代码;
}
void g(void)
{
    //函数代码;
}
```

该程序中出现的 a 和 b 是对外部变量的定义，即给 a 和 b 分配相应的内存空间。在需要使用 a 和 b 的其他文件中的 extern int a，b 是对外部变量的声明，表示本文件引用其他文件中定义的 a 和 b。

外部变量的定义仅能出现一次，而声明可以在不同的文件中出现多次。

5.1.4 流程控制关键字

为了保证程序书写的清晰、规范，建议对于选择、分支、循环结构的执行部分，即使只有一行语句，也把执行部分包围在一对大括号中，这样可以减少逻辑上出错的可能。

1. if、else 关键字

使用 if 关键字时，经常会有判断两个表达式是否相等的关系运算，这时要注意不要把“= =”写成“=”。在某些情况下，把“= =”写成“=”会导致 if 后的条件永远成立，从而失去选择判断功能。例如：

```
int i = 100;
if( i = 1)
{
    printf("OK\n");
}
```

可以看到，由于 if 后的判断语句中，误把“==”写成了“=”，导致条件总是成立，if 下的语句必定被执行，从而失去了选择功能。

if 语句的后面不要无意中加上分号。虽然 C 语言中的语句遇到“;”表示语句的结束，但并不意味着每一条语句的结束都必定要有一个分号，if 语句和 for、while 语句的末尾一般不加分号(除非无选择的执行或者循环体是空语句)。例如，以下程序片段：

```
if( a == b );
```



```
{
f();
}
```

编写者的本意是希望 a 和 b 相等的时候，执行函数 f( )，但是现在却变成了无论 a 和 b 是否相等，f( )都会被执行，原因在于 if 语句的后面加了一个“;”之后，“;”作为 if 语句的执行体被执行了，而真正的 f( )变成了选择语句的下面一条语句，无论 if 语句中的条件是否成立，f( )函数都会被执行。

2．for、while、do…while 关键字

(1) 利用循环关键字构成任务代码

嵌入式系统中的任务一般情况下是以一个无限循环的形式出现的，在没有操作系统支持的单片机系统中，主程序完成系统硬件、软件的初始化后，要做的工作就是在一个无限循环中不停地进行检测、处理；在有实时操作系统支持的情况下，每个任务的主体也都是以无限循环的形式出现，多个任务在操作系统的调度下轮番运行。上述两种情况下的无限循环，一般以下面两种形式出现：

```
while( 1 )
{
    任务代码;
}
```

或者

```
for( ; ;)
{
    任务代码;
}
```


(2) 循环的效率

效率问题从来都是程序设计追求的目标之一，对于运算能力有限的嵌入式系统来说，更要在软件编写的过程中提高效率。

在程序中，如果出现多重循环，只要有可能，应该尽量把循环次数多的循环放在内层，循环次数少的循环放在外层，这样可以减少 CPU 在两层循环间切换的次数，提高程序的执行效率。例如：

```
for( i = 0; i < 10; ++i )
{
    for( j = 0; j < 10000; ++j )
    {
        s = s+a[i][j];
    }
}
```

的效率要较

```
for( i = 0; i< 10000; ++i )
{
for( j = 0; j < 10; ++j )
{
        s = s+a[j][i];
    }
}
```

的效率高。因为第一段程序循环被打断进行切换的次数少。

3．break、continue、goto 关键字

break 关键字用在循环体或者 switch 语句中，用来彻底地跳出循环或者 switch 语句，如同其字面意思，“打破”当前所处的位置，彻底退出。continue 关键字也用在循环语句之中，但是它的作用是先结束本次循环，然后“继续”下一次循环。break 处在多重循环中时，它只能用来退出一重循环，不能退出多重循环，如果要退出多重循环，只有在每层循环中都使用一个 break。例如：

```
for( ; ; )
{
    for( ; ; )
    {
        if( cond1 == 1 )
        {
            flag = 1;
            break;
        }
    }
  if( flag == 1 )
  {
  break;
  }
}
```

在结构化的程序设计中，应该减少 goto 关键字的使用，但并不是彻底的不用 goto。在驱动程序以及其他一些要求程序执行效率较高的场合，可以使用 goto 退出多重循环。

5.1.5　底层系统相关关键字

1．sizeof

对于 sizeof 来说，很多人误以为它是函数，实际上 sizeof 是 C 语言的一个关键字，同时还是一个运算符。sizeof 用来计算一种类型或者一个变量所占的存储空间的字节数。使用形式有如下几种：



```
int i,len;
len = sizeof( int );      //第一种使用形式
len = sizeof( i );        //第二种使用形式
len = sizeof i;           //第三种使用形式
```

sizeof 后如果是数据类型，则数据类型关键字必须放在小括号中；如果是变量，变量既可以放在小括号中，也可以直接跟在 sizeof 关键字后。

使用 sizeof 得出的是数据类型或者变量所占的字节数，在某个平台下编写程序时，如果对某种数据类型的位长不清楚，可以用 sizeof 关键字测试。但是需要注意的是，在有些平台下，一个字节是 16 位，而不是通常的 8 位，例如，前面曾提到的 TI 的 TMSC32x/C2xx/C5x 系列 DSP。

2. volatile

很多编程人员可能对 volatile 不是很熟悉，volatile 的字面意思是“不稳定的”，但是在 C 语言中使用它时恰恰却是希望稳定数据的值。编译器在对源代码进行编译时会进行相应优化，如有一段代码：

```
int i = 1;
int j = i;
int k = i;
```

编译器在给 j、k 赋值时，会认为 i 的值没有发生变化，从而从存放 i 的寄存器中直接取 i 的值赋给 j 和 k。这在一个单线程且没有涉及硬件操作的环境下是正确的。但是如果 i 的值取自某一个硬件端口的数据寄存器或者 i 被其他的线程同时使用，那么就有可能从 CPU 寄存器取出的 i 并不是变量 i 的最后的值。在这种情况下，为了保证 CPU 每次都从内存中取得变量，可以在变量的前面加上一个 volatile 关键字，以防止编译器对变量的优化。

5.1.6　C 语言运算符

C 语言一共有 34 个运算符，运算符的优先级一共有 15 级。表 5-4 详细列出了 34 个运算符的含义，优先级及结合方向。

表 5-4　C 语言的运算符

优先级	运算符	名称或含义	使用形式	结合方向	说明
1	[]	数组下标	数组名[常量表达式]	左到右	
	()	圆括号	(表达式)/函数名(形参表)		
	.	成员选择(对象)	对象.成员名		
	->	成员选择(指针)	对象指针->成员名		
2	-	负号运算符	-表达式	右到左	单目运算符
	(类型)	强制类型转换	(数据类型)表达式		
	++	自增运算符	++变量名/变量名++		单目运算符
	--	自减运算符	--变量名/变量名--		单目运算符

续表

优先级	运算符	名称或含义	使用形式	结合方向	说明
2	*	求目标运算符	*指针变量		单目运算符
	&	求地址运算符	&变量名		单目运算符
	!	逻辑非运算符	!表达式		单目运算符
	～	按位取反运算符	～表达式		单目运算符
	sizeof	长度运算符	sizeof(表达式)		
3	/	除	表达式/表达式	左到右	双目运算符
	*	乘	表达式*表达式		双目运算符
	%	余数(取模)	整型表达式/整型表达式		双目运算符
4	+	加	表达式+表达式	左到右	双目运算符
	-	减	表达式-表达式		双目运算符
5	<<	左移	变量<<表达式	左到右	双目运算符
	>>	右移	变量>>表达式		双目运算符
6	>	大于	表达式>表达式	左到右	双目运算符
	>=	大于等于	表达式>=表达式		双目运算符
	<	小于	表达式<表达式		双目运算符
	<=	小于等于	表达式<=表达式		双目运算符
7	==	等于	表达式==表达式	左到右	双目运算符
	!=	不等于	表达式!=表达式		双目运算符
8	&	按位与	表达式&表达式	左到右	双目运算符
9	^	按位异或	表达式^表达式	左到右	双目运算符
10	|	按位或	表达式|表达式	左到右	双目运算符
11	&&	逻辑与	表达式&&表达式	左到右	双目运算符
12	||	逻辑或	表达式||表达式	左到右	双目运算符
13	?:	条件运算符	表达式 1? 表达式 2: 表达式 3	右到左	三目运算符
14	=	赋值运算符	变量=表达式	右到左	
	/=	除后赋值	变量/=表达式		
	*=	乘后赋值	变量*=表达式		
	%=	取模后赋值	变量%=表达式		
	+=	加后赋值	变量+=表达式		
	-=	减后赋值	变量-=表达式		
	<<=	左移后赋值	变量<<=表达式		
	>>=	右移后赋值	变量>>=表达式		
	&=	按位与后赋值	变量&=表达式		
	^=	按位异或后赋值	变量^=表达式		
	|=	按位或后赋值	变量|=表达式		
15	,	逗号运算符	表达式,表达式,…	左到右	从左向右顺序运算



1. 优先级

关于运算符的含义和使用方法本书不再叙述。重点讲解运算符优先级的记忆方法以及使用中容易出错的地方。

把两个部分构成一个整体的运算符，取整体中某个成员的运算符优先级最高。这类运算符是[](由数组名和下标求数组中的元素)，()(由函数名和参数构成函数)，.(由结构体求其成员)，->(由结构指针求结构体中的成员)。

单目运算符的优先级高于所有的双目运算符和三目运算符。在单目运算符中出现的()是强制类型转换运算符，在语句 int i = (int) f(void)中，包围 int 的()的优先级是第二级，f(void)中的()的优先级是第一级。

C 语言中，一共有三种逻辑运算：逻辑非“!”、逻辑与“&&”、逻辑或“||”。单目逻辑运算符“!”的优先级最高，两个双目逻辑运算“&&”和“||”的优先级低于“!”，其中“&&”的优先级(11 级)又高于“||”的优先级(12 级)。

双目和多目运算符的优先级，按照从高到低的顺序，大致可以这样记忆：算术运算>移位运算>关系运算>位运算(位取反除外)>逻辑运算(逻辑非除外)>条件运算>(复合)赋值运算>逗号运算。在算术运算符中，乘、除、取余的优先级高于加和减，这些读者都比较了解。但是以下几点需要注意：在关系运算中，“>(>=)”、“<(<=)”的优先级高于“==”和“!=”；几种位运算的优先级也是不一样的：位与的优先级高于位或。

在程序编写过程中，容易出错的几个优先级使用有以下几点。

1) “==”和“!=”的优先级高于位运算优先级。val1 & val2 != val3 的正确意义是 val1 & (val2 != val3)，而不是(val1 & val2) != val3。

2) “==”和“!=”的优先级高于赋值运算优先级。ch = getchar() != EOF 的正确意义是 ch = (getchar() != EOF)，而非(ch = getchar())!= EOF。出现上述理解错误的原因是想当然地把表面上类似的两个运算符“!=”和“=”的优先级认为是一样的，实际上，它们的优先级相差很大。

3) 算术运算符优先级高于移位运算符优先级。a << b+c 表示 a <<(b+c)，而非(a << b)+c。

出现上述几种理解错误的原因是，把早已建立并且比较熟悉的法则运用到了相对不太熟悉的运算中，属于思维上的一种惰性，软件设计是一项高度严谨的工作，对于不是太清晰的地方不要想当然。

2. 位运算符的使用

C 语言的位运算一共有六种，按照优先级从高到低依次是：位反“～”、左移“<<”、右移“>>”、位与“&”、位异或“^”、位或“|”。不同于 PC 上的软件开发，在嵌入式软件的设计中，经常会用到对寄存器和内存单元的操作，还有很多的通信协议需要实现，在对寄存器的操作以及通信协议的实现中，会用到大量的位操作，因此，下面介绍一些关于位操作的使用方法和技巧。

首先定义几个整型变量，并且假设 int 型的数据位长是 32 位。

```
int a,b,c,d,e;
```

1) 把整数中的第 n 位置 1：

```
a= a | ( 1 << n );
```

在程序中进行置 1 的操作，可以定义一个带参数的宏：

```
#define SET_BIT(x,n)  ( x = x | ( 1 << n ) )
```

2) 把整数中的第 n 位置 0：

```
a = a &( ~( 1 << n) );
```

在程序中进行置 0 的操作，可以定义一个带参数的宏：

```
#define CLR_BIT(x,n) (x = x & ( ~( 1 << n ))
```

3) 把整数中的第 n 位反转：

```
a = a ^ ( 1 << n );
```

在程序中进行把某位反转的操作，可以定义一个带参数的宏：

```
#define REVERSE_BIT(x,n) (x = x ^ ( 1 << n )
```

4) 测试整数中的第 n 位是否为 1(0)：

```
b = a & ( 1 << n );
```

b 的值等于 1 说明被测整数的第 n 位是 1，b 的值是 0 说明被测整数的第 n 位是 0。如果要测试整数的某位是否为 1，可以定义一个带参数的宏：

```
#define TEST_BIT ( (x & ( 1 << n )) != 0)
```

5) 把整数左移 n 位：

```
a <<= n;
```

把一个整数左移 n 位等价于把该数乘以 2^n。

6) 把整数右移 n 位：

```
a >>= n;
```

把一个整数右移 n 位等价于把该数除以 2^n。

5.2 C 语言的函数

C 语言诞生于 20 世纪 70 年代，当时的软件规模不是太大，面向对象的编程思想尚未提出，C 语言的模块化和面向过程编程的特点决定了函数是组成 C 程序的基本单元。使用 C 语言编写程序，实际上就是编写一个个函数完成任务中各个子任务的功能。

传统的 C 语言程序是由一个或多个函数组成的，每个函数具有相对独立的功能，整个程序功能的实现是由主函数 main()调用其他函数完成的。



5.2.1　函数、变量的定义和声明

1．函数的定义和声明

函数是完成特定功能的一段代码，这段代码被封装起来，具有规定的输入，规定的输出，可以一次编写，多次使用。在程序中使用函数时，必须具有函数的实体。函数按照来源不同，分为库函数和用户自定义函数两类，库函数由编译器或者第三方以二进制代码的形式提供，自定义函数由用户自己编程实现。

函数的定义是对函数具体功能的规定和实现，包括确定函数的参数个数、参数数据类型、函数返回值的数据类型，定义何种数据结构，采用何种算法实现函数的具体功能等。函数的声明则是把函数的名字、函数类型以及形参类型、个数和顺序通知编译器，以便在调用函数时编译器按此进行对照检查(如函数名是否正确，实参与形参的类型和个数是否一致)。从程序中可以看到对函数的声明和函数定义中的函数首部基本上是相同的。

下面给出函数声明和定义的具体例子：

```
//函数声明
int add(int a,int b);
```

函数声明的特征是在函数首部加一个分号“;”。

```
//函数定义
int add(int a,int b)
{
        return a+b;
}
```

函数定义的特征是函数首部后面没有分号，其下有一对大括号，大括号中是函数功能的具体实现。

函数的声明可以出现在程序的任意位置，在所有函数的外部或者函数的内部都可以，而函数的定义只能出现在其他函数的外部，不允许在函数的内部进行。

函数的定义给编译器提供产生可执行二进制代码的原料，声明是告诉编译器关于被调用函数的特征，声明的作用主要是在程序的编译阶段对调用函数的合法性进行全面检查。在 C 语言中，函数声明也称为函数原型(Function Prototype)。使用函数原型是 ANSI C 的一个重要特点。

实际上，如果在函数调用前，没有对函数进行声明，则编译器会把第一次遇到的该函数形式(函数定义或函数调用)作为函数的声明，并将函数类型默认为是 int 型。例如，有一个 add()函数，调用之前没有进行函数声明，编译时首先遇到的函数形式是函数调用“add(a,b)”，由于对原型的处理是不考虑参数名的，因此系统将 add(a,b)加上 int 作为函数声明，即 int add()。所以很多书上说，如果函数返回值类型为整型，可以在函数调用前不作函数声明。但是使用这种方法时，系统无法对参数的类型做检查。即使调用函数时参数使用不当，在编译时也不会报错。因此，为了程序清晰和安全，建议都添加声明。

如果被调用函数的定义出现在主调函数之前，可以不必声明。因为编译系统已经先知道了已定义的函数类型，会根据函数首部提供的信息对函数的调用做正确性检查。

在进行程序设计时，一般的做法是把函数的声明写在一个头文件中，然后在所有可能调用这些函数的文件中包含含有函数声明的头文件。

2. 变量的定义和声明

在程序中，函数的定义和声明比较容易区分，而变量的声明和定义就显得稍微复杂一点。变量的声明有两种情况。

1) 需要建立存储空间的(定义、声明)。例如，int a 在声明的时候就已经建立了存储空间。

2) 不需要建立存储空间的(声明)。例如，extern int a，其中变量 a 是在别的文件中定义的。

前者是“定义性声明(Defining Declaration)”或者称为“定义(Definition)”，而后者是“引用性声明(Referencing Declaration)”。从广义的角度来讲声明中包含着定义，但是并非所有的声明都是定义，例如，int a 既是声明，同时又是定义。然而对于 extern int a 来讲它只是声明不是定义。一般情况下，把建立空间的声明称之为“定义”，把不需要建立存储空间的声明称之为“声明”。很明显一般所说的声明范围是比较窄的，也就是说非定义性质的声明。

```
int main()
{
    extern int A;          //这是个声明而不是定义,声明 A 是一个已经定义了的外部变量
    do_sth();
}
int A;                     //这是定义,定义了 A 为整型的外部变量(全局变量)
```


外部变量(全局变量)的“定义”与外部变量的“声明”是不相同的,外部变量的定义只能有一次,它的位置是在所有函数之外,而同一个文件中的外部变量声明可以是多次的,它可以在函数之内(哪个函数要用就在哪个函数中声明)，也可以在函数之外(在外部变量的定义点之前)。系统会根据外部变量的定义(而不是根据外部变量的声明)分配存储空间。对于外部变量来讲，初始化只能在“定义”中进行，而不能在“声明”中进行。所谓的“声明”，其作用是声明该变量是一个已在后面定义过的外部变量，仅仅是为了“提前”引用该变量而作的“声明”而已。extern 只作声明，不作定义。

5.2.2 变量的作用域和生命期

变量的作用域和生命期是 C 语言中的一个重要概念，特别是在碰到局部变量和全局变量同名，函数的形参和实参同名的情况，更会让人感到疑惑。本节将以表格的形式总结变量作用域和生命期的特点。

如表 5-5 所示，作用域有三种。



表 5-5　变量的作用域和生命期

<table>
<tr><th>变量类型</th><th>生命期</th><th>作用域</th></tr>
<tr><td rowspan="3">局部变量</td><td>auto：自动变量，离开定义函数即消失</td><td rowspan="3">只作用于该函数内部</td></tr>
<tr><td>register：寄存器变量，离开定义函数即消失</td></tr>
<tr><td>static：变量，离开定义函数仍存在</td></tr>
<tr><td rowspan="3">全局变量</td><td rowspan="3">在程序运行期间，一直存在</td><td>static：仅限于本文件内部调用</td></tr>
<tr><td>extern：若未使用 static 修饰，当别的文件调用此变量时，加 extern 修饰表明是外部引用</td></tr>
<tr><td>注：从变量定义(引用)处起，至程序结束一直有效</td></tr>
</table>

注：函数本身即为全局的，若仅限于本文件调用，则应用 static 修饰

extern(外部的)：这是在函数外部定义的变量的默认存储方式。extern 变量的作用域是整个程序。

static(静态的)：在函数外部说明为 static 的变量的作用域为从定义点到该文件尾部；在函数内部说明为 static 的变量的作用域为从定义点到该局部程序块尾部。

auto(自动的)：这是在函数内部说明的变量的缺省存储方式。auto 变量的作用域为从定义点到该局部程序块尾部。

变量的生命期也有三种，但它们不像作用域那样有预定义的关键字名称。

第一种是 extern 和 static 变量的生存期，它们从 main()函数被调用之前开始，到程序退出时为止。

第二种是函数参数和 auto 变量的生存期，它们从函数调用时开始，到函数返回时为止。

第三种是动态分配的数据的生存期，它们从程序调用 malloc()或 calloc()为数据分配存储空间时开始，到程序调用 free()或程序退出时为止。

5.2.3　函数间的参数传递

一个程序中，不同的函数间为了相互协作，完成某个功能，一般需要相互传递数据，C 语言中，函数间的数据传递可以使用参数、返回值、全局变量进行。

函数调用发生时，函数间参数的传递遵循“按值传递，单向传递”的原则。假设有一个函数 f 定义如下：

```
int f(int x,int y)
{
   do_sth();
}
```

当在函数 g()中调用 f()时：

```
void g()
{
    int a,b ;
```

```
        f(a,b) ;
    }
```

C 语言规定参数的传递是“按值传递”，即当函数调用发生时，系统给形参分配存储空间，然后把实参的值传递到形参的存储空间。在被调用函数中对形参进行处理，实际上是在形参所占的存储空间中进行的，对形参的改变不会作用到实参的存储空间，即不会影响到实参。函数调用完毕后，在函数中对形参的改变不会反馈给实参(上例中，在函数 f()未被调用之前，系统没有给形参分配存储空间，因此形参也就没有值可言。当一个函数被调用后，系统给形参分配相应的存储空间，把实参的值复制到分配给形参的存储空间中。在被调用函数中，对形参的任何改变都和实参没有关系，并且被调用函数执行完毕后，形参值的改变也不会反馈到调用函数的实参中)。这个过程可以用图 5.4 表示。

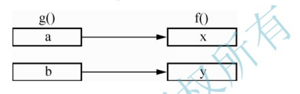


图 5.4　实参和形参之间值的传递

为了在被调用函数中对实参进行修改，必须采取地址传送方式传递数据。这时作为参数传递的不是数据本身，而是数据的存储地址。在这种方式中，以数据的地址作为实参调用一个函数，而被调用函数的形参必须是可以接收地址值的指针变量，并且它的数据类型要与被传递数据的数据类型相同。下面给出一个使用地址传送方式传递数据的例子：

```
/*地址传送方式传递数据*/
int plus(int *px,int *py)
{
    int z;
    z = *px+*py;
    return z;
}

main()
{
    int a,b,c;
    printf("Enter A and B:");
    scanf("%d %d",&a,&b);
    c = plus(&a,&b);
    printf("A+B=%d\n",c);
}
```

使用地址传送方式传递数据的特点：由于数据无论是在调用函数中还是被调用函数中都使用同一个存储空间，所以在被调用函数中对该存储空间中的值做出某种变动后，必然会影响到使用该空间的调用函数中相应变量的值。地址传送方式的特点如图 5.5 所示。





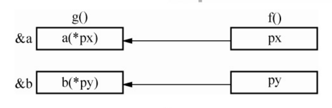


图 5.5　函数间的地址传送

在 C 语言中使用地址传送参数，不仅可以传送变量的地址，而且可以传送数组或者其他构造类型的地址。特别需要指出的是，C 语言中如果使用数组名字作为参数，实参传递给形参的是数组首元素的首地址，而不是把所有数组元素的值传递给形参。在被调用函数中对数组元素值的改变其实就是改变调用函数中的数组元素，修改可以反馈回调用函数中，因为调用函数和被调用函数之间互相传递的是数组元素的地址。这种情况，可以用下面代码验证。

```
/*数组作为函数参数，实参传递给形参的是数组首元素的首地址*/
main()
{ void swap(int x[2]);
  int a[2]={2,9};
  swap(a);
  printf("%d,%d\n",a[0],a[1]);
}
void  swap(int x[2])
{ int t;
   t=x[0];x[0]=x[1];x[1]=t;
}
```


5.2.4　利用参数返回结果

利用参数传送地址时，在被调用函数中对形参的存储空间中的数据做出的任何变动，都会反馈给调用函数中作为实参的相应变量。利用这个特性，可以在被调用函数中把它的处理结果送入某个形参的存储空间。当函数返回时，通过该形参的存储空间把处理结果带给调用函数。

```
 /*利用参数返回处理结果*/
plus(int x,int y, int *z)
{
      *z = x+y;
}
void main()
{
    int a,b,c;
    printf("Enter A and B:");
    scanf("%d %d",&a,&b);
    plus(a,b,&c);
    printf("A+B=%d\n",c);
}
```

在上例中，x 和 y 用于接收两个运算数，形参 z 是一个指针，它的作用是把相加的结果送入它所指向的存储地址中。在主函数中调用函数 plus()时，第三个实参是&c，即变量 c 的地址。根据地址传送方式的特性可知，调用 plus()后，变量 c 中得到了 a 和 b 的相加结果。

在 C 语言中，关键字 return 只能返回一个结果，如果被调用函数中有多个结果需要返回给调用者，可以使用参数进行返回。下面的例子中，调用函数 cal()后，会把参数 x 和 y 的和差积商同时返回。

```
/*利用参数返回多个处理结果*/
cal(int x,int y, int *a, int *b, int *c, int *d)
{
      *a = x+y;
      *b = x - y;
      *c = x *y;
      *d = x / y;
}
main()
{
    int a,b,c;
    printf("Enter A and B:");
    scanf("%d %d",&a,&b);
    plus(a,b,&c);
    printf("A+B=%d,A-B=%d,A * B =%d,A / B =%d\n",a,b,c,d);
}
```


5.3 预 处 理

C 语言中的预处理包括文件包含、宏定义、条件编译等。预处理功能是在程序的预编译阶段进行的替换等操作，而不是由编译器直接对预处理部分进行编译。预处理语句与普通的 C 语句的区别在于，预处理语句以“#”开头，且末尾不带分号。本节将讨论宏定义和条件编译。

5.3.1 宏定义

宏定义的一般形式是

```
#define <标识符> <字符串>
```

其含义是将程序中出现<标识符>的地方全部利用<字符串>来替换。使用宏定义时，需要注意括号的使用。例如，定义一个宏用来求 x 的平方：

```
#define SQUARE(x) x*x
```

如果用宏求 10 的平方，则 SQUARE(10)会被替换成 10*10，结果正确。但是如果用宏



定义求(10+2)的平方，则 SQUARE(10+2)会被替换成 10+2*10+2，显然结果是错误的。重新定义宏如下：

```
#define SQUARE(x) (x)*(x)
```

用上面新定义的宏求 SQUARE(10+2)，结果是正确的。下面再定义一个宏：

```
#define SUM(x) (x)+(x)
```

用宏求 SUM(1*2)* SUM(1*2)，替换的结果是(1*2)+ (1*2)*(1*2)+ (1*2)。这不是希望的结果，因此把上面的宏定义字符串部分再加上括号改写如下：

```
#define SUM(x) ((x)+(x))
```

则 SUM(1*2)* SUM(1*2)的替换就正确了。因此，在使用宏定义时，在逻辑正确的前提下，多使用括号是一种确保替换正确的手段。

通过上面的例子，可以看出一些简单的功能，既可以使用函数，也可以使用宏替换来实现，那么究竟是用函数好，还是宏定义好？这就要求对二者进行合理的取舍。下面来看一个例子，比较两个数或者表达式的大小，首先把它写成宏定义：

```
#define MAX( a, b) ( (a) > (b) (a) : (b) )
```

其次，把它用函数来实现：

```
int max( int a, int b)
{
    return (a > b a : b);
}
```

很显然，我们不会选择用函数来完成这个任务，原因有两个。第一，函数调用会带来额外的开销，它需要开辟一片栈空间，记录返回地址，将形参压栈，从函数返回还要释放栈。这种开销不仅会降低代码效率，而且代码量也会大大增加，使用宏定义则在代码规模和速度方面都比函数更胜一筹。第二，函数的参数必须被声明为一种特定的类型，所以它只能在类型合适的表达式上使用，如果要比较两个浮点型的大小，就不得不再写一个专门针对浮点型的比较函数。反之，上面的那个宏定义可以用于整形、长整形、单精度浮点型、双精度浮点型以及其他任何可以用“>”操作符比较值大小的类型，也就是说，宏是与类型无关的。

和使用函数相比，使用宏的不利之处在于每次使用宏时，一份宏定义代码的副本都会插入到程序中。除非宏非常短，否则使用宏会大幅度增加程序的长度。还有一些任务根本无法用函数实现，但是用宏定义却比较容易。例如，参数类型没法作为参数传递给函数，但是可以把参数类型传递给带参的宏。看下面的例子：

```
#define MALLOC(n, type) ( (type *) malloc((n)* sizeof(type)))
```

利用这个宏，可以为任何类型分配一段指定大小的空间，并返回指向这段空间的指针。下面观察一下这个宏的工作过程：

```
int *ptr;
ptr = MALLOC ( 5, int );
```

将这个宏展开以后的结果：

```
ptr = (int *) malloc ( (5) * sizeof(int) );
```

这个例子是宏定义的经典应用之一，完成了函数不能完成的功能，但是宏定义也不能滥用，通常，如果相同的代码需要出现在程序的几个地方，更好的方法是把它实现为一个函数。表 5-6 是对宏和函数不同之处的总结。

表 5-6　宏和函数的比较

属性	#define 宏	函数
代码长度	每次使用时，宏代码都被插入到程序中。除了非常小的宏之外，程序的长度将大幅度增长	函数代码只出现于一个地方：每次使用这个函数时，都调用那个地方的同一份代码
执行速度	更快	存在函数调用、返回的额外开销
操作符优先级	宏参数的求值是在所有周围表达式的上下文环境里，除非它们加上括号，否则邻近操作符的优先级可能产生不可预料的结果	函数参数只在函数调用时求值一次，它的结果值传递给函数。表达式的求值结果更容易预测
参数求值	参数用于宏定义时，每次都将重新求值，由于多次求值，具有副作用的参数可能会产生不可预测的结果	参数在函数调用前只求值一次，在函数中多次使用参数并不会导致多次求值过程，参数的副作用并不会造成任何特殊问题
参数类型	宏与类型无关，只要参数的操作是合法的，它可以用于任何参数类型	函数的参数是与类型有关系的，如果参数的类型不同，就需要使用不同的函数，即使它们执行的任务是相同的

5.3.2　条件编译

条件编译，顾名思义就是对满足条件的部分进行编译。条件编译一共有三种形式：

```
/*第一种类型的条件编译*/
#ifdef 标识符
        程序段 1
#else
        程序段 2
#endif
```

第一种条件编译的作用是如果标识符已经被 #define 宏定义，在编译时只编译程序段 1，否则编译程序段 2。其中，#else 部分也可以没有。

```
/*第二种类型的条件编译*/
#ifndef 标识符
        程序段 1
#else
```



```
        程序段 2
#endif
```

第二种条件编译的作用是如果标识符没有被 #define 宏定义，在编译时只编译程序段 1，否则编译程序段 2。其中，#else 部分也可以没有。

```
/*第三种类型的条件编译*/
#if 表达式
        程序段 1
#else
        程序段 2
#endif
```

第三种条件编译的作用是如果表达式的值为真(非零)时就编译程序段 1，表达式的值为零时，就编译程序段 2。

利用条件编译可以对程序进行非常彻底并且随心所欲的注释。当在程序中已经有了很多注释的情况下，无论是再次使用多行注释“/* */”或者单行注释“//”，编译时往往都会报很多错误，把以前的注释删掉重新添加会比较麻烦，在这种情况下就可以把要加注释的程序段用条件编译 #if 0 和 #endif 包围起来，例如，有一段已经事先加了很多注释的代码，现在用条件编译进行彻底的注释：

```
#if 0
 int a,b,c,d;
 /*

*/
    //unsigned char a[4]={0x42,0x0e,0xa5,0x00};
    float f;
    //f = asicc_2_float(a);
    //printf("%f\n", f);

    unsigned char b[4]={0x42,0x55,0xd0,0x00};
    //printf("Please input 4 asicc :");
    //scanf("%d %d %d %d ",&a[0],&a[1],&a[2],&a[3]);
    f = asicc_2_float(b);
    printf("%f\n", f);//
#endif
```

在上面例子中，由于 #if 后面的条件为假，因此编译器对 #if 0 和 #endif 之间的代码不予编译，这样也就起到了多行注释的效果。这种用法在实际工作中较为有用，望读者加以注意。

5.4 指　　针

指针是 C 语言中最复杂，最难掌握的内容，也是 C 语言中功能最为强大的工具。可以说没有学会指针的使用，也就没有掌握 C 语言。

本节将对指针的本质、指针的赋值与初始化、指针和数组的关系、指针数组和数组指针、函数指针和指针函数、直接向内存写入数值等方面进行深入的讨论。

5.4.1 指针的本质

程序运行时，常量、变量、函数等都会在内存中占据一定的存储空间，任何一处存储空间都有一个表示其位置的编号——地址。如果知道了变量的存储地址就可以读写变量，知道了函数的存储地址就可以调用函数。在 C 语言中，用指针来表示地址。指针就是存放地址值的变量或者常量。指针分为指针常量和指针变量，通常把指针变量简称为指针。

指针变量定义的一般形式：

```
<数据类型标识符>  *<标识符>
```

指针变量就其本质来说是一个变量，它在内存中也占据一定的存储空间，但是这个变量不同于用自身的值直接参加计算的普通变量，指针变量表示的是其他变量、函数、常量或者指针变量的地址。定义指针变量时的数据类型并不是指针变量本身的数据类型，而是指针变量所指向的目标变量的数据类型。实际上在同一个系统下，所有的指针变量具有相同的数据类型，无论它所指向的目标变量是什么。例如，在 32 位系统下，所有的指针变量的长度都是 4 个字节，这可以用 sizeof(void *)进行测试。下面看一个例子：

```
int *a;  //语句1
char *c; //语句2
```

语句 1 定义了一个指向 int 型数据的指针变量，这个指针的名字是 a，它的值是一个 int 型数的存储地址，a 本身在内存中有自己的存储空间，在 32 位系统下占据 4 个字节。语句 2 定义了一个指向 char 型数据的指针变量，这个变量的名字是 c，它的值是一个 char 型数据的存储地址，c 本身在内存中有自己的存储空间，在 32 位系统下占据 4 个字节。虽然指针变量(指向)的目标的数据类型可以有多种多样，但是在同一个系统中，指针变量自身的数据类型都是一样的。

5.4.2 指针的赋值与初始化

指针作为一种特殊的变量，既可以在使用的过程中被赋予不同的值，也可以在定义的同时就赋一个初值，即初始化。下面是给指针赋值和初始化的例子：

```
int a,b;
int *p ;
p = &a;                  //给指针赋值
int *q = &b;             //对指针初始化
```



定义一个指针变量，仅仅是给这个指针变量分配存储空间，指针变量所占存储空间中的数据即指针变量的值是不确定的，也就是指针变量没有明确的指向一个目标变量，在这种情况下对指针变量进行取目标计算，因为指针变量的值可能是一个非法地址或者受系统保护的地址，有可能引起程序运行出错。所以指针在使用前一定要进行赋值。例如：

```
int *p = 0;
```

这一行代码的意思是定义一个指针变量，它的名字是 p，它所占的内存空间中存放的是一个 int 型数据的地址，在定义指针变量的同时把它的值设为 0，0 是作为初值赋给指针变量本身的。再如：

```
int *p;      //语句 1
*p = 0;      //语句 2
```

第一句的意思是定义一个指向 int 型数据的指针变量，第二句的意思是把 0 赋给 p 所指向的目标变量，而非指针变量。因此 int *p = 0 和*p = 0 是完全不同的。

上述对整型指针变量的使用比较清晰易懂。下面再看字符指针的初始化和赋值。在对字符指针进行初始化时，可以这样来写 char *s = “ABCD”。在程序中也可以用类似 char *s;s = “ABCD”的形式对字符指针进行赋值。char *s = “ABCD”的意思是定义一个指向字符型数据的指针 s，s 的值是字符串“ABCD”中第一个字符的地址。这里赋值符号“=”表示的是把字符串中第一个字符的地址赋给指针变量 s，而非把整个字符串的值赋给 s。在把一个字符串的首地址赋给字符指针时，赋值号的左边是指针变量，右边是字符串，并不是对指针变量进行取目标运算后再赋值。

5.4.3　指针和数组

指针是用来存储地址的一种特殊变量，在同一个系统下，无论指针变量的目标是什么，指针变量的长度都是一样的，这在前面已经讨论过。数组是若干个某种数据类型的数据的有序集合，数组在内存中占据一片连续的存储空间。数组所占的存储空间的大小取决于数组元素的数据类型和元素个数。

数组在内存中存储，每个元素都有一个地址，数组中第一个元素的地址作为数组的地址。既然数组也有地址，那么就可以把数组的地址存入指针变量中，也即指针变量指向数组。这就是指针和数组之间的关系。

如果定义了一个数组 int a[10]，则 a 是一个地址常量，表示数组中首元素的首地址，而非很多教材中所说的 a 是数组的首地址。数组的首地址其实应该是 &a。关于这一点，可以用下面的一段程序来验证。

```
main()
{
    int a[5] = {1,2,3,4,5};
    int *p1,*p2;
    int (*pa)[5];
    p1 = a ;
```

```
        p2 =(int *) &a ;//&a 是数组的首地址
        pa = &a;
        printf("p1 = 0X%0X,p2 = 0X%0X,pa = 0X%0X\n",p1,p2,pa);

        p1 = a+1;
        p2 =(int *) (&a+1); //&a+1 指向下一个数组的首地址
        pa = &a;
        printf("p1 = 0X%0X,p2 = 0X%0X,pa = 0X%0X\n",p1,p2,pa);
    }
```

在 PC 上的 VC++ 6.0 环境下，上面程序运行输出的结果：

```
p1=0X12FF6C,p2= 0X12FF6C,pa = 0X12FF6C
p1=0X12FF70,p2= 0X12FF80,pa = 0X12FF6C
```

显然，a 作为数组首元素的首地址和 &a 作为数组的首地址在数值上是一样的，但是对这两个地址分别进行加一运算后，结果就不同了，a+1 变成 0X12FF70，是数组中下一个元素的地址，而 &a+1 变成 0X12FF80，是下一个数组的地址。另外，如果把 p2 =(int *) &a 这一句中的强制类型转换(int *)去掉，编译时会出错，编译器给出的提示："error C2440：'='：cannot convert from 'int (*)[5]' to 'int *'"(错误 C2440：'='：不能从 'int (*)[5]' 转换成 'int *')。int (*)[5]是一个数组指针，int *是一个整型指针，出错的原因是不能把一个数组指针的值赋给一个整型指针，从这个错误提示也可以看出 &a才是数组的首地址。

在弄清楚了数组的地址，数组首元素的地址后，下面讨论数组元素的表示方法。

在 C 语言中，数组元素一共有三种不同的表示方法：下标法，指针常量法和指针变量法。如果定义一个数组 int a[5] = {1,2,3,4,5}，则三种方法表示如下。

1. 下标法 a[i]

这是最常用的一种表示数组中元素的方法。在 C 语言中，对以下标进行操作的数组元素是转化成指针进行处理的。例如，进行 a[4]这个操作时这样进行：a 是数组中首元素的首地址，首地址加上 4 个元素的偏移量，即 a+4 * d，计算出新元素的地址，从新的地址中取出数据，即 a[4]。可以这样理解：a[4]=*(a+4)= *(4+a)= 4[a] 。为了验证上面的说法，读者可以这样进行测试：在定义了数组 a 之后，用 4[a]来取数组中的元素，观察得到的结果是否和 a[4]一样。

2. 指针常量法*(a+i)

前面讲过，数组的名字是数组首元素的首地址，数组中每个元素的存储地址等于数组名字加上元素的序号，这样可以用常量表达数组中每个元素的存储地址，有了地址之后，再对已知地址进行取目标运算，就可以取得相应的数组元素。例如：

```
        int i;
        for( i= 0; i <5; ++i)
        {
            printf("%d\n",*(a+i));
        }
```






3．指针变量法*(p+i)

p 是和数组元素具有相同数据类型的一个指针变量，通过赋值运算使 p 指向数组中的一个元素。该方法和方法 2 的不同之处是，把数组中某个元素的地址赋给一个指针变量，通过对指针变量的改变获取其他元素的地址，再对新地址进行取目标运算以求得数组元素的值。例如：

```
for( p = a;p < a+10; ++p)
{
    printf("%d\n",*p);
}
```

无论使用哪种方法表示数组元素，实质上都是根据数组首元素的首地址和数组元素的序号计算出要进行操作的数组元素的地址，再对该地址处进行取目标运算而得到元素的值。即 a[i]等价于*(a+i)，*(p+i)等价于 p[i]，下标可以认为是相对于某个基地址的元素个数的偏移量，这样，当用下标法表示数组元素时，下标不但可以是 0，还可以是负数。请读者验证下面的例子：

```
p=&a[4];
printf("p[-1] = %d,p[-2] = %d\n",p[-1],p[-2]);
```

5.4.4　指针数组和数组指针

指针数组，是这样一个数组：数组中的元素是指针，指针所指向的目标的数据类型由定义指针数组时的数据类型给出。例如，int *p1[10]定义了一个指针数组，“[]”的优先级比“*”的优先级高，p1 首先和“[]”相结合构成一个数组，名字是 p1，int *表示数组元素的类型。所以上述定义表达的意思是定义一个数组，名字是 p1，数组中有 10 个元素，每个元素都是一个指向 int 型数据的指针。指针数组就是存储指针的数组。

数组指针是一个指针，只不过这个指针的目标不是一个普通变量，而是一个数组。例如，int (*p2)[10]定义了一个指针，在这里“()”的优先级比“[]”的优先级高，因此 p2 是一个指针，int 表示的是数组的内容，所以指针 p2 的目标是一个具有 10 个元素的 int 型数组。数组指针就是指向数组的指针。下面给出一个指针数组和数组指针的程序实例：

```
#include <stdio.h>
main()
{
static int m[3][4]={0,1,2,3,4,5,6,7,8,9,10,11};//定义二维数组 m 并初始化
int  (*p)[4];      //数组指针 p 是指针，指向一维数组，每个一维数组有 4 个 int 型元素
int i,j;
int *q[3];         //指针数组 q 是数组，数组元素是指针，3 个 int 型指针
p=m;               //p 是指针，可以直接指向二维数组
printf("--数组指针输出元素--/n");
for(i=0;i<3;i++)//输出二维数组中各个元素的数值      {
    for(j=0;j<4;j++)
```

```
        {
           printf("%3d ",*(*(p+i)+j));
        }
           printf("\n");
         }
        printf("\n");
        for(i=0;i<3;i++,p++)                    //p 可看成是行指针
        {
            printf("%3d ",**p);                 //每一行的第一个元素
            printf("%3d ",*(*p+1));             //每一行的第二个元素
            printf("%3d ",*(*p+2));             //每一行的第三个元素
            printf("%3d ",*(*p+3));             //每一行的第四个元素
            printf("\n");
        }
        printf("\n");
        printf("--指针数组输出元素--\n");
        for(i=0;i<3;i++)
            q[i]=m[i];                          //q 是数组，元素 q[i] 是指针
        for(i=0;i<3;i++)
        {
            for(j=0;j<4;j++)
            {
                printf("%3d ",q[i][j]);//q[i][j]可换成*(q[i]+j)
            }
            printf("\n");
        }
        printf("\n");
        q[0]=m[0];
        for(i=0;i<3;i++)
        {
            for(j=0;j<4;j++)
            {
                printf("%3d ",*(q[0]+j+4*i));
            }
            printf("\n");
        }
        printf("\n");
    }
```


5.4.5 函数指针和指针函数

请看下面两个声明：

```
int *f(int a,int b);  //声明1
int (*f)(int a,int b);//声明2
```





这两个声明看起来非常相像，但是含义截然不同：声明 1 声明了一个指针函数；声明 2 声明了一个函数指针。

指针是一种特殊的变量，指针变量的值是程序组成单元的地址，程序的组成单元可以是各种常量、变量或者函数。如果一个指针变量的值是某个函数的地址，即指针指向一个函数，那么这个指针就是函数指针。对函数指针进行取目标计算“*”就是调用函数指针指向的函数。函数指针定义的一般形式：

```
<类型> (*函数指针)(函数的参数列表);
```

下面是几个函数指针定义的例子：

```
void (*p)();//定义一个函数指针p，p指向一个输入参数和返回值均是void的函数
int (*pf)(int a,int b);/*定义一个函数指针 pf，pf 指向一个函数，这个函数有两个
int 型的参数，返回值的数据类型也是 int 型*/
```

数组的名字代表数组中第一个元素的首地址，函数的名字表示函数代码存储的起始地址，在给函数指针赋值时，只需将函数的名字赋给函数指针即可。例如：

```
int func(int x);          //定义一个函数 func
int (*pf)(int x);         //定义一个函数指针
pf = func;                //把函数 func 的首地址赋给函数指针 pf
```

给函数指针赋值时，函数名称后面不带括号，也不带参数。

如果 pi 是一个指向 int 型数据 i 的指针，则*pi 就等于 i；同样，如果 pfunc 是一个指向函数 func 的指针，那么*pfunc 就是调用函数 func。下面是通过函数指针调用函数的例子：

```
#include<stdio.h>
#include<string.h>
char* fun(char* p1,char* p2)
{
    int i =0;
    i =strcmp(p1,p2);
    if (0 == i)
    {
    return p1;
    }
    else
    {
    return p2;
    }
}
int main()
{
    char* (*pf)(char* p1,char* p2); //语句1
    pf= &fun;                       //语句2
```

```
    (*pf)("aa","bb");                    //语句3
return 0;
}
```

上面程序中的(*pf)(“aa”，“bb”)语句就是通过函数指针调用函数。从上例中可以看到，通过函数指针调用函数通常包括三个步骤：首先定义一个函数指针(语句 1)，然后给函数指针赋值(语句 2)，即让函数指针指向一个具体的函数，最后通过对函数指针的取目标运算实现对函数的调用(语句 3)。

既然可以直接调用一个函数使其运行，那么为什么又引入函数指针，通过函数指针来调用函数呢？这是因为调用者在调用某个函数时，每次需要完成的操作不是固定的，这时使用函数指针指向不同函数的地址即可。

指针函数是一种函数，这种函数的返回值是一个指针变量，这个指针变量可以指向一个变量或者是函数。指针函数声明的一般形式：

```
类型名  *函数名(参数表);
```

例如：

```
int *add(int a,int b);
```

上面声明了一个函数 add( )，add 具有两个整形参数 a 和 b，add 的返回值是一个指针，这个指针指向一个 int 型数。在 add 的两侧分别是“*”和“( )”运算符，“( )”的优先级高于“*”，因此 add 先和“( )”结合，构成一个函数，然后再和“*”结合，表示函数的返回值是一个指针变量。最前面的 int 表示函数返回的指针指向一个 int 型变量。下面给出一个指针函数的例子：

```
/*将字符串1(str1)复制到字符串2(str2)，并输出字符串2*/
#include "stdio.h"
main()
{
    char *ch(char *, char *);
    char str1[]="I am glad to meet you!";
    char str2[]="Welcom to study C!";
    printf("%s", ch(str1, str2));
}
char *ch(char *str1, char *str2)
{
    int i;
    char *p;
    p=str2;
    if(*str2==NULL)  exit(-1);
    do
    {
        *str2=*str1;
```



```
            str1++;
            str2++;
        }while(*str1!=NULL);
        return(p);
    }
```

由上面的讲述可以看出，函数指针是一个指针，这个指针指向一个函数；指针函数是一个函数，这个函数的返回值是一个指针，二者有明显的区别。

5.4.6　直接向内存写入数值

在嵌入式软件的编程中，有时会碰到需要向一个指定地址写入一个值的操作，例如，基于 ARM 内核的微处理器，外设寄存器和内存统一编址，当要对外设进行操作时，实际上就是向一个指定的地址写入指定值。

现在假设向地址 0x3000 开始的地址写入一个值 0x100，可以用下面的代码实现：

```
int *p = (int *)0x3000;
*p = 0x100;
```

在地址 0x3000 前需要使用一个强制类型转换，告诉编译器这个地址中存放的是 int 型数据。如果不定义指针，也可以这样写：

```
*(int *)0x3000 = 0x100;
```

在嵌入式系统的开发中，比较常用的做法是通过宏定义给指定的地址起一个名字，作为变量来使用，在程序中可以直接给宏赋值。例如：

```
#define REG ( *(int *)0x3000)
REG = 0x100;
```


本 章 小 结

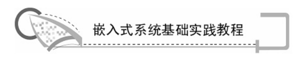
本章针对嵌入式软件开发需要对系统底层进行操作的特点，重点阐述了 C 语言中一些容易出错和需要透彻理解的内容。

(1) C 语言的关键字和运算符：分别介绍了容易出错的四类关键字的使用方法，重点强调了在嵌入式软件开发中，数据类型的位长、字节顺序、浮点数和零值的比较等，介绍了 volatile 关键字的使用方法，纠正了 sizeof 被很多人误认为是函数的错误观念。

(2) C 语言的函数：分析了函数、变量定义和声明的区别，回顾了各种变量的作用域和生命期，讲解了 C 语言中参数传递的规则，并介绍了利用指针使一个函数返回多个结果的方法。

(3) 预处理：重点分析了宏定义使用难点，并给出了解决办法，回顾了条件编译的三种形式，并给出一种利用条件编译实现完全彻底注释程序的方法。

(4) 指针：首先透彻分析了指针的本质，继而讨论了指针变量的赋值和初始化，指出



了指针和数组的关系，然后讲解了指针数组和数组指针、指针函数和函数指针两组较难的概念，并给出了程序实例，最后介绍了通过指针直接向内存写入数值的方法。



阅读材料

TIOBE 编程语言排行榜

当前的编程语言有上百种之多，当然不是每种语言的使用机会都均等。初学者总是想知道哪些语言用的最多最广泛，以便有针对性地去学习，因为付出同样大的代价学习的编程语言用途很小是任何人都不愿接受的。国外网站 TIOBE(www.tiobe.com)专门做编程语言热度的排行，非常专业并且客观。

TIOBE 公司成立于 2000 年 10 月 1 日，由瑞士公司 Synspace 和一些独立的投资人创建。TIOBE 是“The Importance Of Being Earnest”的缩写。该公司主要关注于软件质量的评估。TIOBE 程序设计语言指数是由该公司推出并进行维护的，这个指数将程序设计语言以排名列表的形式提供出来，并且每个月更新一次，用来表示程序设计语言的流行度。TIOBE 编程语言排行榜的网址是 http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html。

TIOBE 评估是通过统计该编程语言在主流搜索引擎上被搜索的次数来计算的。搜索包括在搜索引擎、新闻组及博客上的搜索等。主流搜索引擎由 Alexa.com 网站上的排名来决定。如果用“hits(PL#i,SE)”表示编程语言 PL 在搜索引擎 SE 上的指数排名为 i 的搜索次数，n 表示搜索引擎个数，则 PL 在前 50 名编程语言中排名评估的计算公式为((hits(PL#i,SE1)/hits(PL#1)+⋯+hits(PL#50))+⋯+(hits(PL#i,SEn)/hits(PL#1)+⋯+hits(PL#50)))/n。

除了排名的评估方法以外，编程语言的状态也是该指数的一个组成部分。状态主要分为两种，A 表示主流语言，B 表示非主流语言。另外还有 A-和 A--用来表示 A 和 B 两个状态的中间状态。

如果一个编程语言在过去三个月的评估中有一次的评估超过 0.7%，该语言就可以得到一个 A，否则状态为 B。另外，关于 A-和 A--，如果过去三个月中有两次评估超过 0.7%，则状态为 A-；如果过去三个月中有一个评估超过 0.7%，则状态为 A--。

表 5-7 是摘自 TIOBE 网站的 2012 年 8 月排名前 20 的编程语言排行(http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html)。

表 5-7　TIOBE 网站 2012 年 8 月排名前 20 的编程语言

Position Aug 2012	Position Aug 2011	Delta in Position	Programming Language	Ratings Aug 2012	Delta Aug 2011	Status
1	2	↑	C	18.937%	+1.55%	A
2	1	↓	Java	16.352%	−3.06%	A
3	6	↑↑↑	Objective-C	9.540%	+4.05%	A
4	3	↓	C++	9.333%	+0.90%	A
5	5	=	C#	6.590%	+0.55%	A
6	4	↓↓	PHP	5.524%	−0.61%	A
7	7	=	(Visual) Basic	5.334%	+0.32%	A
8	8	=	Python	3.876%	+0.46%	A



续表

Position Aug 2012	Position Aug 2011	Delta in Position	Programming Language	Ratings Aug 2012	Delta Aug 2011	Status
9	9	=	Perl	2.273%	−0.04%	A
10	12	↑↑	Ruby	1.691%	+0.36%	A
11	10	↓	JavaScript	1.365%	−0.19%	A
12	13	↑	Delphi/Object Pascal	1.012%	−0.06%	A
13	14	↑	Lisp	0.975%	+0.07%	A
14	26	↑↑↑↑↑↑↑↑↑↑	Visual Basic .NET	0.877%	+0.41%	A
15	15	=	Transact-SQL	0.849%	+0.03%	A
16	18	↑↑	Pascal	0.793%	+0.13%	A
17	11	↓↓↓↓↓↓	Lua	0.726%	−0.64%	A--
18	16	↓↓	Ada	0.649%	−0.05%	B
19	22	↑↑↑	PL/SQL	0.610%	+0.08%	B
20	29	↑↑↑↑↑↑↑↑↑	MATLAB	0.533%	+0.09%	B

习　　题

一、选择题

1．设有如下定义语句：

int m[ ]={2,4,6,8},*k=m;

以下选项中，表达式的值为 6 的是(　　)。

　A．*(k+2)　　　B．k+2　　　C．*k+2　　　D．*k+=2

2．viod fun(double a[], int *n)

{……}

以下叙述中正确的是(　　)。

　A．调用 fun 函数时只有数组执行按值传送，其他实参和形参之间执行按地址传送

　B．形参 a 和 n 都是指针变量

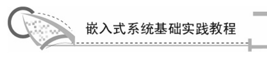
　C．形参 a 是一个数组名，n 是指针变量

　D．调用 fun 函数时将把 double 型实参数组元素一一对应地传送给形参 a 数组

二、判断题

1．指针只能指向变量或者常量，不能指向函数。　　　　　　　　　　　　　　(　　)

2．C 语言中数组作为函数的参数时，按照“按值传递”的原则，实参数组的元素全部复制到形参数组中。　　　　　　　　　　　　　　　　　　　　　　　　(　　)



三、简答题

请说出下列表达式的含义

```
int *a;
int **a;
int *a[10];
int (*a)[10];
int  *a(int);
int  (*a)(int);
```

四、程序设计题

1. 编写程序测试你所用的计算机系统的大小端格式。
2. 编写程序验证 sizeof 关键字的用法。
3. 编写程序计算一个 32 位整数中 0 和 1 的个数。
4. 编写一个宏，求出所给的两个数中值较小的一个。
5. 编写程序，自定义一个数组，用三种方法输出数组中的元素。
6. 写出两种向指定地址写入指定值的程序片段。

